Package 'stepR'

October 18, 2024

Title Multiscale Change-Point Inference

Version 2.1-10

Depends R (>= 3.3.0)

Imports Rcpp (>= 0.12.3), lowpassFilter (>= 1.0.0), R.cache (>= 0.10.0), digest (>= 0.6.10), stats, graphics, methods

LinkingTo Rcpp

Suggests testthat (>= 1.0.0), knitr

VignetteBuilder knitr

Description Allows fitting of step-functions to univariate serial data where neither the number of jumps nor their positions is known by implementing the multiscale regression estimators SMUCE, simulataneous multiscale changepoint estimator, (K. Frick, A. Munk and H. Sieling, 2014) <doi:10.1111/rssb.12047> and HSMUCE, heterogeneous SMUCE, (F. Pein, H. Sieling and A. Munk, 2017) <doi:10.1111/rssb.12202>. In addition, confidence intervals for the change-point locations and bands for the unknown signal can be obtained.

License GPL-3

Classification/MSC 62G08, 92C40, 92D20

LazyData yes

NeedsCompilation yes

Author Pein Florian [aut, cre],
Thomas Hotz [aut],
Hannes Sieling [aut],
Timo Aspelmeier [ctb]

Maintainer Pein Florian <f.pein@lancaster.ac.uk>

Repository CRAN

Date/Publication 2024-10-18 11:00:03 UTC

Contents

stepR-package 2
BesselPolynomial 8
bounds 9

compareBlocks . . . 10
computeBounds . . . 12
computeStat . . . 15
contMC . . . 17
critVal . . . 19
dfilter . . . 26
family . . . 28
intervalSystem . . . 29
jsmurf . . . 30
jumpint . . . 32
monteCarloSimulation . . . 34
MRC . . . 36
MRC.1000 . . . 39
MRC.asymptotic . . . 40
MRC.asymptotic.dyadic . . . 40
neighbours . . . 41
parametricFamily . . . 41
penalty . . . 44
sdrobnorm . . . 46
smuceR . . . 47
stepblock . . . 49
stepbound . . . 51
stepcand . . . 52
stepFit . . . 54
stepfit . . . 57
steppath . . . 60
stepsel . . . 62
testSmallScales . . . 63
transit . . . 65

Index **67**

`stepR-package` *Multiscale Change-Point Inference*

Description

Allows fitting of step-functions to univariate serial data where neither the number of jumps nor their positions is known by implementing the multiscale regression estimators SMUCE (*Frick et al.*, 2014) and HSMUCE (*Pein et al.*, 2017). In addition, confidence intervals for the change-point locations and bands for the unknown signal can be obtained. This is implemented in the function `stepFit`. Moreover, technical quantities like the statistics itself, bounds or critical values can be computed by other functions of the package. More details can be found in the vignette.

Details

New in version 2.0-0:

stepFit	Piecewise constant multiscale inference
critVal	Critical values
computeBounds	Computation of the bounds
computeStat	Computation of the multiscale statistic
monteCarloSimulation	Monte Carlo simulation
parametricFamily	Parametric families
intervalSystem	Interval systems
penalty	Penalties

From version 1.0-0:

compareBlocks	Compare fit blockwise with ground truth
neighbours	Neighbouring integers
sdrobnorm	Robust standard deviation estimate
stepblock	Step function
stepcand	Forward selection of candidate jumps
stepfit	Fitted step function
steppath	Solution path of step-functions
stepsel	Automatic selection of number of jumps

Mainly used for patchclamp recordings and may be transferred to a specialised package:

BesselPolynomial	Bessel Polynomials
contMC	Continuous time Markov chain
dfilter	Digital filters
jsmurf	Reconstruct filtered piecewise constant functions with noise
transit	TRANSIT algorithm for detecting jumps

Deprecated (please read the documentation of them theirself for more details):

MRC	Compute Multiresolution Criterion
MRC.1000	Values of the MRC statistic for 1,000 observations (all intervals)
MRC.asymptotic	"Asymptotic" values of the MRC statistic (all intervals)
MRC.asymptotic.dyadic	"Asymptotic" values of the MRC statistic(dyadic intervals)
bounds	Bounds based on MRC
family	Family of distributions
smuceR	Piecewise constant regression with SMUCE

Storing of Monte-Carlo simulations

If `q == NULL` in `critVal`, `stepFit` or `computeBounds` a Monte-Carlo simulation is required for computing critical values. Since a Monte-Carlo simulation lasts potentially much longer (up to several hours or days if the number of observations is in the millions) than the main calculations, this package offers multiple possibilities for saving and loading the simulations. Simulations can either be saved in the workspace in the variable `critValStepRTab` or persistently on

the file system for which the package R.cache is used. Moreover, storing in and loading from variables and RDS files is supported. Finally, a pre-simulated collection of simulations can be accessed by installing the package stepRdata available from http://www.stochastik.math.uni-goettingen.de/stepRdata_1.0-0.tar.gz. By default simulations will be saved in the workspace and on the file system. For more details and for how simulation can be removed see Section *Simulating, saving and loading of Monte-Carlo simulations* in critVal.

References

Frick, K., Munk, A., Sieling, H. (2014) Multiscale change-point inference. With discussion and rejoinder by the authors. *Journal of the Royal Statistical Society, Series B* **76**(3), 495–580.

Pein, F., Sieling, H., Munk, A. (2017) Heterogeneous change point inference. *Journal of the Royal Statistical Society, Series B*, **79**(4), 1207–1227.

Pein, F., Tecuapetla-Gómez, I., Schütte, O., Steinem, C., Munk, A. (2017) Fully-automatic multiresolution idealization for filtered ion channel recordings: flickering event detection. *arXiv*:1706.03671.

Hotz, T., Schütte, O., Sieling, H., Polupanow, T., Diederichsen, U., Steinem, C., and Munk, A. (2013) Idealizing ion channel recordings by a jump segmentation multiresolution filter. *IEEE Transactions on NanoBioscience* **12**(4), 376–386.

VanDongen, A. M. J. (1996) A new algorithm for idealizing single ion channel data containing multiple unknown conductance levels. *Biophysical Journal* **70**(3), 1303–1315.

Futschik, A., Hotz, T., Munk, A., Sieling, H. (2014) Multiresolution DNA partitioning: statistical evidence for segments. *Bioinformatics*, **30**(16), 2255–2262.

Boysen, L., Kempe, A., Liebscher, V., Munk, A., Wittich, O. (2009) Consistencies and rates of convergence of jump-penalized least squares estimators. *The Annals of Statistics* **37**(1), 157–183.

Davies, P. L., Kovac, A. (2001) Local extremes, runs, strings and multiresolution. *The Annals of Statistics* **29**, 1–65.

Friedrich, F., Kempe, A., Liebscher, V., Winkler, G. (2008) Complexity penalized M-estimation: fast computation. *Journal of Computational and Graphical Statistics* **17**(1), 201–224.

See Also

stepFit, critVal, computeStat, computeBounds, jsmurf, transit, sdrobnorm, compareBlocks, parametricFamily, intervalSystem, penalty

Examples

```
# generate random observations
set.seed(1)
n <- 100L
x <- seq(1 / n, 1, 1 / n)
mu <- stepfit(cost = 0, family = "gauss", value = c(0, 3, 0, -2, 0), param = NULL,
              leftEnd = x[c(1, 21, 26, 71, 81)],
              rightEnd = x[c(20, 25, 70, 80, 100)], x0 = 0,
              leftIndex = c(1, 21, 26, 71, 81),
              rightIndex = c(20, 25, 70, 80, 100))
sigma0 <- 0.5
epsilon <- rnorm(n, 0, sigma0)
```

```
y <- fitted(mu) + epsilon
plot(x, y, pch = 16, col = "grey30", ylim = c(-3, 4))
lines(mu, lwd = 3)

# computation of SMUCE and its confidence statements
fit <- stepFit(y, x = x, alpha = 0.5, jumpint = TRUE, confband = TRUE)
lines(fit, lwd = 3, col = "red", lty = "22")

# confidence intervals for the change-point locations
points(jumpint(fit), col = "red", lwd = 3)
# confidence band
lines(confband(fit), lty = "22", col = "darkred", lwd = 2)

# higher significance level for larger detection power, but less confidence
# suggested for screening purposes
stepFit(y, x = x, alpha = 0.9, jumpint = TRUE, confband = TRUE)

# smaller significance level for the small risk that the number of
# change-points is overestimated with probability not more than 5%,
# but smaller detection power
stepFit(y, x = x, alpha = 0.05, jumpint = TRUE, confband = TRUE)

# different interval system, lengths, penalty and given parameter sd
stepFit(y, x = x, alpha = 0.5, intervalSystem = "dyaLen",
        lengths = c(1L, 2L, 4L, 8L), penalty = "weights",
        weights = c(0.4, 0.3, 0.2, 0.1), sd = sigma0,
        jumpint = TRUE, confband = TRUE)

# the above calls saved and (attempted to) load Monte-Carlo simulations and
# simulated them for nq = 128 observations
# in the following call no saving, no loading and simulation for n = 100
# observations is required, progress of the simulation will be reported
stepFit(y, x = x, alpha = 0.5, jumpint = TRUE, confband = TRUE, messages = 1000L,
        options = list(simulation = "vector", load = list(), save = list()))

# critVal was called in stepFit, can be called explicitly,
# for instance outside of a for loop to save computation time
qVector <- critVal(100L, alpha = 0.5)
identical(stepFit(y, x = x, q = qVector, jumpint = TRUE, confband = TRUE), fit)

qValue <- critVal(100L, alpha = 0.5, output = "value")
identical(stepFit(y, x = x, q = qValue, jumpint = TRUE, confband = TRUE), fit)

# computeBounds gives the multiscale contraint
computeBounds(y, alpha = 0.5)

# monteCarloSimulation will be called in critVal if required
# can be called explicitly
stat <- monteCarloSimulation(n = 100L)
identical(critVal(n = 100L, alpha = 0.5, stat = stat),
          critVal(n = 100L, alpha = 0.5,
                  options = list(load = list(), simulation = "vector")))
identical(critVal(n = 100L, alpha = 0.5, stat = stat, output = "value"),
```

```
          critVal(n = 100L, alpha = 0.5, output = "value",
                  options = list(load = list(), simulation = "vector")))

stat <- monteCarloSimulation(n = 100L, output = "maximum")
identical(critVal(n = 100L, alpha = 0.5, stat = stat),
          critVal(n = 100L, alpha = 0.5,
                  options = list(load = list(), simulation = "vector")))
identical(critVal(n = 100L, alpha = 0.5, stat = stat, output = "value"),
          critVal(n = 100L, alpha = 0.5, output = "value",
                  options = list(load = list(), simulation = "vector")))

# fit satisfies the multiscale contraint, i.e.
# the computed penalized multiscale statistic is not larger than the global quantile
computeStat(y, signal = fit, output = "maximum") <= qValue
# multiscale vector of statistics is componentwise not larger than
# the vector of critical values
all(computeStat(y, signal = fit, output = "vector") <= qVector)


# family "hsmuce"
set.seed(1)
y <- c(rnorm(50, 0, 1), rnorm(50, 1, 0.2))
plot(x, y, pch = 16, col = "grey30", ylim = c(-2.5, 2))

# computation of HSMUCE and its confidence statements
fit <- stepFit(y, x = x, alpha = 0.5, family = "hsmuce",
               jumpint = TRUE, confband = TRUE)
lines(fit, lwd = 3, col = "red", lty = "22")

# confidence intervals for the change-point locations
points(jumpint(fit), col = "red", lwd = 3)
# confidence band
lines(confband(fit), lty = "22", col = "darkred", lwd = 2)

# for comparison SMUCE, not recommend to use here
lines(stepFit(y, x = x, alpha = 0.5,
              jumpint = TRUE, confband = TRUE),
      lwd = 3, col = "blue", lty = "22")


# family "mDependentPS"
# generate observations from a moving average process
set.seed(1)
y <- c(rep(0, 50), rep(2, 50)) +
  as.numeric(arima.sim(n = 100, list(ar = c(), ma = c(0.8, 0.5, 0.3)), sd = sigma0))
correlations <- as.numeric(ARMAacf(ar = c(), ma = c(0.8, 0.5, 0.3), lag.max = 3))
covariances <- sigma0^2 * correlations
plot(x, y, pch = 16, col = "grey30", ylim = c(-2, 4))

# computation of SMUCE for dependent observations with given covariances
fit <- stepFit(y, x = x, alpha = 0.5, family = "mDependentPS",
               covariances = covariances, jumpint = TRUE, confband = TRUE)
lines(fit, lwd = 3, col = "red", lty = "22")
```

```
# confidence intervals for the change-point locations
points(jumpint(fit), col = "red", lwd = 3)
# confidence band
lines(confband(fit), lty = "22", col = "darkred", lwd = 2)

# for comparison SMUCE for independent observations, not recommend to use here
lines(stepFit(y, x = x, alpha = 0.5,
              jumpint = TRUE, confband = TRUE),
      lwd = 3, col = "blue", lty = "22")

# with given correlations, standard deviation will be estimated by sdrobnorm
stepFit(y, x = x, alpha = 0.5, family = "mDependentPS",
        correlations = correlations, jumpint = TRUE, confband = TRUE)


# examples from version 1.0-0
# estimating step-functions with Gaussian white noise added
# simulate a Gaussian hidden Markov model of length 1000 with 2 states
# with identical transition rates 0.01, and signal-to-noise ratio 2
sim <- contMC(1e3, 0:1, matrix(c(0, 0.01, 0.01, 0), 2), param=1/2)
plot(sim$data, cex = 0.1)
lines(sim$cont, col="red")
# maximum-likelihood estimation under multiresolution constraints
fit.MRC <- smuceR(sim$data$y, sim$data$x)
lines(fit.MRC, col="blue")
# choose number of jumps using BIC
path <- steppath(sim$data$y, sim$data$x, max.blocks=1e2)
fit.BIC <- path[[stepsel.BIC(path)]]
lines(fit.BIC, col="green3", lty = 2)

# estimate after filtering
# simulate filtered ion channel recording with two states
set.seed(9)
# sampling rate 10 kHz
sampling <- 1e4
# tenfold oversampling
over <- 10
# 1 kHz 4-pole Bessel-filter, adjusted for oversampling
cutoff <- 1e3
df.over <- dfilter("bessel", list(pole=4, cutoff=cutoff / sampling / over))
# two states, leaving state 1 at 10 Hz, state 2 at 20 Hz
rates <- rbind(c(0, 10), c(20, 0))
# simulate 0.5 s, level 0 corresponds to state 1, level 1 to state 2
# noise level is 0.3 after filtering
Sim <- contMC(0.5 * sampling, 0:1, rates, sampling=sampling, family="gaussKern",
  param = list(df=df.over, over=over, sd=0.3))
plot(Sim$data, pch = ".")
lines(Sim$discr, col = "red")
# fit under multiresolution constraints using filter corresponding to sample rate
df <- dfilter("bessel", list(pole=4, cutoff=cutoff / sampling))
Fit.MRC <- jsmurf(Sim$data$y, Sim$data$x, param=df, r=1e2)
lines(Fit.MRC, col = "blue")
```

```
# fit using TRANSIT
Fit.trans <- transit(Sim$data$y, Sim$data$x)
lines(Fit.trans, col = "green3", lty=2)
```

`BesselPolynomial` *Bessel Polynomials*

Description

Recursively compute coefficients of Bessel Polynomials.

Deprecation warning: This function is a help function for the Bessel filters in `dfilter` and may be removed when `dfilter` will be removed.

Usage

```
BesselPolynomial(n, reverse = FALSE)
```

Arguments

`n`	order
`reverse`	whether to return the coefficients of a reverse Bessel Polynomial

Value

Returns the polynom's coefficients ordered increasing with the exponent, i.e. starting with the intercept, as for `polyroot`.

See Also

`dfilter`, `bessel`, `polyroot`

Examples

```
# 15 x^3 + 15 x^2 + 6 x + 1
BesselPolynomial(3)
```

bounds *Bounds based on MRC*

Description

Computes two-sided bounds for a set of intervals based on a multiresolution criterion (MRC).

Deprecation warning: This function is deprecated, but still working, however, may be defunct in a future version. Please use instead the function computeBounds. An example how to reproduce results (currently only family "gauss" is supported) is given below.

Usage

```
bounds(y, type = "MRC", ...)
bounds.MRC(y, q, alpha = 0.05, r = ceiling(50 / min(alpha, 1 - alpha)),
  lengths = if(family == "gaussKern")
    2^(floor(log2(length(y))):ceiling(log2(length(param$kern)))) else
    2^(floor(log2(length(y))):0), penalty = c("none", "len", "var", "sqrt"),
 name = if(family == "gaussKern") ".MRC.ktable" else ".MRC.table", pos = .MCstepR,
 family = c("gauss", "gaussvar", "poisson", "binomial","gaussKern"), param = NULL,
  subset, max.iter = 1e2, eps = 1e-3)
## S3 method for class 'bounds'
x[subset]
```

Arguments

y	a numeric vector containing the serial data
type	so far only bounds of type "MRC" are implemented
...	further arguments to be passed on to bounds.MRC
q	quantile of the MRC; if specified, alpha and r will be ignored
alpha	level of significance
r	number of simulations to use to obtain quantile of MRC for specified alpha
lengths	vector of interval lengths to use, dyadic intervals by default
penalty	penalty term in the multiresolution statistic: "none" for no penalty, "len" for penalizing the length of an interval, "var" for penalizing the variance over an interval, and "sqrt" for penalizing the square root of the MRC
family, param	specifies distribution of data, see family
subset	a subset of indices of y for which bounds should be aggregated
name, pos	under which name and where precomputed results are stored, or retrieved, see assign
max.iter	maximal iterations in Newton's method to compute non-Gaussian MRC bounds
eps	tolerance in Newton's method
x	an object of class bounds

Value

Returns an object of class bounds, i.e. a list whose entry bounds contains two-sided bounds (lower and upper) of the considered intervals (with left index li and right index ri) in a data.frame, along with a vector start specifying in which row of entry bounds intervals with corresponding li start (if any; specified as a C-style index), and a logical feasible telling whether a feasible solution exists for these bounds (always TRUE for MRC bounds which are not restricted to a subset).

See Also

computeBounds, stepbound, family

Examples

```
y <- rnorm(100, c(rep(0, 50), rep(1, 50)), 0.5)
b <- computeBounds(y, q = 4, intervalSystem = "dyaLen", penalty = "none")
b <- b[order(b$li, b$ri), ]
attr(b, "row.names") <- seq(along = b$li)

# entries in bounds are recovered by computeBounds
all.equal(bounds(y, q = 4)$bounds, b) # TRUE


# simulate signal of 100 data points
Y <- rpois(100, 1:100 / 10)
# compute bounds for intervals of dyadic lengths
b <- bounds(Y, penalty="len", family="poisson", q=4)
# compute bounds for all intervals
b <- bounds(Y, penalty="len", family="poisson", q=4, lengths=1:100)
```

---

compareBlocks *Compare fit blockwise with ground truth*

Description

Blockwise comparison of a fitted step function with a known ground truth using different criteria.

Usage

```
compareBlocks(truth, estimate, dist = 5e3)
```

Arguments

truth	an object of class stepblock giving the ground truth, or a list of such objects
estimate	corresponding estimated object(s) of class stepblock
dist	a single numeric specifying the distance for at which jumps will be considered as having matched in the qualitative criterion

Value

A data.frame, containing just one row if two single stepblock were given, with columns

true.num, est.num
: the true / estimated number of blocks

true.pos, false.pos, false.neg, sens.rate, prec.rate
: the number of true / false positive, false negatives, as well as the corresponding sensitivity and precision rates, where an estimated block is considered a true positive if it there is a corresponding block in the ground truth with both end-points within dist of each other

fpsle
: false positive sensitive localization error: for each estimated block's midpoint find into which true block it falls, and sum distances of the respective borders

fnsle
: false negative sensitive localization error: for each true block's mid-point find into which estimated block it falls, and sum distances of the respective borders

total.le
: total localization error: sum of fpsle and fnsle

Note

No differences between true and fitted parameter *values* are taking into account, only the precision of the detected blocks is considered; also, differing from the criteria in Elhaik et al.~(2010), no blocks are merged in the ground truth if its parameter values are close, as this may punish sensitive estimators.

Beware that these criteria compare *blockwise*, i.e. they do *not* compare the precision of single jumps but for each block both endpoints have to match well at the same time.

References

Elhaik, E., Graur, D., Josić, K. (2010) Comparative testing of DNA segmentation algorithms using benchmark simulations. *Molecular Biology and Evolution* **27**(5), 1015-24.

Futschik, A., Hotz, T., Munk, A. Sieling, H. (2014) Multiresolution DNA partitioning: statistical evidence for segments. *Bioinformatics*, **30**(16), 2255–2262.

See Also

stepblock, stepfit, contMC

Examples

```
# simulate two Gaussian hidden Markov models of length 1000 with 2 states each
# with identical transition rates being 0.01 and 0.05, resp, signal-to-noise ratio is 5
sim <- lapply(c(0.01, 0.05), function(rate)
  contMC(1e3, 0:1, matrix(c(0, rate, rate, 0), 2), param=1/5))
plot(sim[[1]]$data)
lines(sim[[1]]$cont, col="red")
# use smuceR to estimate fit
fit <- lapply(sim, function(s) smuceR(s$data$y, s$data$x))
lines(fit[[1]], col="blue")
# compare fit with (discretised) ground truth
compareBlocks(lapply(sim, function(s) s$discr), fit)
```

`computeBounds` *Computation of the bounds*

Description

Computes the multiscale contraint given by the multiscale test, (3.12) in the vignette. In more detail, returns the bounds of the interval of parameters for which the test statistic is smaller than or equal to the critical value for the corresponding length, i.e. the two solutions resulting from equating the test statistic to the critical value.
If `q == NULL` a Monte-Carlo simulation is required for computing critical values. Since a Monte-Carlo simulation lasts potentially much longer (up to several hours or days if the number of observations is in the millions) than the main calculations, this package saves them by default in the workspace and on the file system such that a second call requiring the same Monte-Carlo simulation will be much faster. For more details, in particular to which arguments the Monte-Carlo simulations are specific, see Section *Storing of Monte-Carlo simulations* below. Progress of a Monte-Carlo simulation can be reported by the argument `messages` and the saving can be controlled by the argument `option`, both can be specified in `...` and are explained in `monteCarloSimulation` and `critVal`, respectively.

Usage

```
computeBounds(y, q = NULL, alpha = NULL, family = NULL,
              intervalSystem = NULL, lengths = NULL, ...)
```

Arguments

`y`	a numeric vector containing the observations
`q`	either `NULL`, then the vector of critical values at level `alpha` will be computed from a Monte-Carlo simulation, or a numeric giving the global quantile or a numeric vector giving the vector of critical values. Either `q` or `alpha` must be given. Otherwise, `alpha == 0.5` is chosen with a `warning`. This argument will be passed to `critVal` to obtain the needed critical values. Additional parameters for the computation of `q` can be specified in `...`, for more details see the documentation of `critVal`. Please note that by default the Monte-Carlo simulation will be saved in the workspace and on the file system, for more details see Section *Storing of Monte-Carlo simulations* below
`alpha`	a probability, i.e. a single numeric between 0 and 1, giving the significance level. Its choice is a trade-off between data fit and parsimony of the estimator. In other words, this argument balances the risks of missing change-points and detecting additional artefacts. For more details on this choice see (Frick et al., 2014, section 4) and (Pein et al., 2017, section 3.4). Either `q` or `alpha` must be given. Otherwise, `alpha == 0.5` is chosen with a `warning`
`family`	a string specifying the assumed parametric family, for more details see parametricFamily, currently `"gauss"`, `"hsmuce"` and `"mDependentPS"` are supported. By default (`NULL`) `"gauss"` is assumed

`intervalSystem` a string giving the used interval system, either `"all"` for all intervals, `"dyaLen"` for all intervals of dyadic length or `"dyaPar"` for the dyadic partition, for more details see intervalSystem. By default (`NULL`) the default interval system of the specified parametric family will be used, which one this will be is described in `parametricFamily`

`lengths` an integer vector giving the set of lengths, i.e. only intervals of these lengths will be considered. Note that not all lengths are possible for all interval systems and for all parametric families, see intervalSystem and parametricFamily, respectively, to see which ones are allowed. By default (`NULL`) all lengths that are possible for the specified `intervalSystem` and for the specified parametric family will be used

`...` there are two groups of further arguments:

1. further parameters of the parametric family. Depending on argument `family` some might be required, but others might be optional, please see parametricFamily for more details,
2. further parameters that will be passed to `critVal`. `critVal` will be called automatically with the number of observations `n = length(y)`, the arguments `family`, `intervalSystem`, `lengths`, `q` and `output` set. For these arguments no user interaction is required and possible, all other arguments of `critVal` can be passed additionally

Value

A `data.frame` containing two integer vectors `li` and `ri` and two numeric vectors `lower` and `upper`. For each interval in the set of intervals specified by `intervalSystem` and `lengths` `li` and `ri` give the left and right index of the interval and `lower` and `upper` give the lower and upper bounds for the parameter on the given interval.

Storing of Monte-Carlo simulations

If `q == NULL` a Monte-Carlo simulation is required for computing critical values. Since a Monte-Carlo simulation lasts potentially much longer (up to several hours or days if the number of observations is in the millions) than the main calculations, this package offers multiple possibilities for saving and loading the simulations. Progress of a simulation can be reported by the argument `messages` which can be specified in `...` and is explained in the documentation of `monteCarloSimulation`. Each Monte-Carlo simulation is specific to the number of observations, the parametric family (including certain parameters, see parametricFamily) and the interval system, and for simulations of class `"MCSimulationMaximum"`, additionally, to the set of lengths and the used penalty. Monte-Carlo simulations can also be performed for a (slightly) larger number of observations n_q given in the argument `nq` in `...` and explained in the documentation of `critVal`, which avoids extensive resimulations for only a little bit varying number of observations. Simulations can either be saved in the workspace in the variable `critValStepRTab` or persistently on the file system for which the package `R.cache` is used. Moreover, storing in and loading from variables and RDS files is supported. Finally, a pre-simulated collection of simulations can be accessed by installing the package `stepRdata` available from `http://www.stochastik.math.uni-goettingen.de/stepRdata_1.0-0.tar.gz`. The simulation, saving and loading can be controlled by the argument `option` which can be specified in `...` and is explained in the documentation of `critVal`. By default simulations

will be saved in the workspace and on the file system. For more details and for how simulation can be removed see Section *Simulating, saving and loading of Monte-Carlo simulations* in `critVal`.

Note

Depending on `intervalSystem` and `lengths` the intervals might be ordered differently to allow fast computation. For most applications the order should not matter. Otherwise, the entries can be reordered with `order`, an example is given below.

References

Frick, K., Munk, A., Sieling, H. (2014) Multiscale change-point inference. With discussion and rejoinder by the authors. *Journal of the Royal Statistical Society, Series B* **76**(3), 495–580.

Pein, F., Sieling, H., Munk, A. (2017) Heterogeneous change point inference. *Journal of the Royal Statistical Society, Series B*, **79**(4), 1207–1227.

See Also

`critVal`, `penalty`, `parametricFamily`, `intervalSystem`, `stepFit`, `computeStat`, `monteCarloSimulation`

Examples

```
y <- c(rnorm(50), rnorm(50, 1))

# the multiscale contraint
bounds <- computeBounds(y, alpha = 0.5)

# the order of the bounds depends on intervalSystem and lengths
# to allow fast computation
# if a specific order is required it can be reordered by order
# b is ordered with increasing left indices and increasing right indices
b <- bounds[order(bounds$li, bounds$ri), ]
attr(b, "row.names") <- seq(along = b$li)

# higher significance level for larger detection power, but less confidence
computeBounds(y, alpha = 0.99)

# smaller significance level for stronger confidence statements, but at
# the risk of missing change-points
computeBounds(y, alpha = 0.05)

# different interval system, lengths, penalty and given parameter sd
computeBounds(y, alpha = 0.5, intervalSystem = "dyaLen",
              lengths = c(1L, 2L, 4L, 8L), penalty = "weights",
              weights = c(0.4, 0.3, 0.2, 0.1), sd = 0.5)

# with given q
identical(computeBounds(y, q = critVal(100L, alpha = 0.5)), bounds)
identical(computeBounds(y, q = critVal(100L, alpha = 0.5, output = "value")),
          bounds)
```

```
# the above calls saved and (attempted to) load Monte-Carlo simulations and
# simulated them for nq = 128 observations
# in the following call no saving, no loading and simulation for n = 100
# observations is required, progress of the simulation will be reported
computeBounds(y, alpha = 0.5, messages = 1000L,
              options = list(simulation = "vector",
                             load = list(), save = list()))

# with given stat to compute q
stat <- monteCarloSimulation(n = 128L)
identical(computeBounds(y, alpha = 0.5, stat = stat),
          computeBounds(y, alpha = 0.5, options = list(load = list())))
```

`computeStat` *Computation of the multiscale statistic*

Description

Computes the multiscale vector of penalised statistics, (3.7) in the vignette, or the penalised multiscale statistic, (3.6) in the vignette, for given signal.

Usage

```
computeStat(y, signal = 0, family = NULL, intervalSystem = NULL, lengths = NULL,
            penalty = NULL, nq = length(y),
            output = c("list", "vector", "maximum"), ...)
```

Arguments

`y`	a numeric vector containing the observations
`signal`	the given signal, either a single numeric for a constant function equal to the given value or an object of class `stepfit`. More precisely, a `list` containing an integer vector `leftIndex`, an integer vector `rightIndex` and a numeric vector `value`, all of the same length, e.g. a `data.frame`, specifying a step function is enough
`family`	a string specifying the assumed parametric family, for more details see parametricFamily, currently `"gauss"`, `"hsmuce"` and `"mDependentPS"` are supported. By default (`NULL`) `"gauss"` is assumed
`intervalSystem`	a string giving the used interval system, either `"all"` for all intervals, `"dyaLen"` for all intervals of dyadic length or `"dyaPar"` for the dyadic partition, for more details see intervalSystem. By default (`NULL`) the default interval system of the specified parametric family will be used, which one this will be is described in `parametricFamily`

lengths	an integer vector giving the set of lengths, i.e. only intervals of these lengths will be considered. Note that not all lengths are possible for all interval systems and for all parametric families, see intervalSystem and parametricFamily, respectively, to see which ones are allowed. By default (NULL) all lengths that are possible for the specified intervalSystem and for the specified parametric family will be used
penalty	a string specifying how the statistics will be penalised, either "sqrt", "log" or "none", see penalty and section 3.2 in the vignette for more details. By default (NULL) the default penalty of the specified parametric family will be used, which one this will be is described in parametricFamily
nq	a single integer larger than or equal to length(y) giving the number of observations used in the penalty term, see penalty for more details. The possibility to use a number larger than length(y) is given for comparisons, since a (slightly) larger number can be chosen in critVal and monteCarloSimulation to avoid extensive recomputations for (slightly) varying number of observations. For more details see also the Section *Simulating, saving and loading of Monte-Carlo simulations* in critVal
output	a string specifying the output, see *Value*
...	further parameters of the parametric family. Depending on argument family some might be required, but others might be optional, please see parametricFamily for more details

Value

If output == list a list containing in maximum the penalised multiscale statistic, i.e. the maximum over all test statistics, in stat the multiscale vector of penalised statistics, i.e. a vector of length lengths giving the maximum over all tests of that length, and in lengths the vector of lengths. If output == vector a numeric vector giving the multiscale vector of penalised statistics. If output == maximum a single numeric giving the penalised multiscale statistic. -Inf is returned for lengths for which on all intervals of that length contained in the set of intervals the signal is not constant and, hence, no test statistic can be computed. This behaves similar to max(numeric(0)).

References

Frick, K., Munk, A., Sieling, H. (2014) Multiscale change-point inference. With discussion and rejoinder by the authors. *Journal of the Royal Statistical Society, Series B* **76**(3), 495–580.

Pein, F., Sieling, H., Munk, A. (2017) Heterogeneous change point inference. *Journal of the Royal Statistical Society, Series B*, **79**(4), 1207–1227.

See Also

parametricFamily, intervalSystem, penalty, monteCarloSimulation, stepFit, computeBounds

Examples

```
y <- rnorm(100)
# for the default signal = 0 a signal constant 0 is assumed
```

```
identical(computeStat(y), computeStat(y,
            signal = list(leftIndex = 1L, rightIndex = 100L, value = 0)))

# different constant value
ret <- computeStat(y, signal = 1)
# penalised multiscale statistic
identical(ret$maximum, computeStat(y, signal = 1, output = "maximum"))
# multiscale vector of penalised statistics
identical(ret$stat, computeStat(y, signal = 1, output = "vector"))

y <- c(rnorm(50), rnorm(50, 1))
# true signal
computeStat(y, signal = list(leftIndex = c(1L, 51L), rightIndex = c(50L, 100L),
                             value = c(0, 1)))

# fit satisfies the multiscale contraint, i.e.
# the penalised multiscale statistic is not larger than the used global quantile 1
computeStat(y, signal = stepFit(y, q = 1), output = "maximum") <= 1

# different interval system, lengths, penalty, given parameter sd
# and computed for an increased number of observations nq
computeStat(y, signal = list(leftIndex = c(1L, 51L), rightIndex = c(50L, 100L),
                             value = c(0, 1)), nq = 128, sd = 0.5,
            intervalSystem = "dyaLen", lengths = c(1L, 2L, 4L, 8L), penalty = "none")

# family "hsmuce"
computeStat(y, signal = mean(y), family = "hsmuce")

# family "mDependentPS"
signal <- list(leftIndex = c(1L, 13L), rightIndex = c(12L, 17L), value = c(0, -1))
y <- c(rep(0, 13), rep(-1, 4)) +
  as.numeric(arima.sim(n = 17, list(ar = c(), ma = c(0.8, 0.5, 0.3)), sd = 1))
covariances <- as.numeric(ARMAacf(ar = c(), ma = c(0.8, 0.5, 0.3), lag.max = 3))
computeStat(y, signal = signal, family = "mDependentPS", covariances = covariances)
```

`contMC` *Continuous time Markov chain*

Description

Simulate a continuous time Markov chain.

Deprecation warning: This function is mainly used for patchlamp recordings and may be transferred to a specialised package.

Usage

```
contMC(n, values, rates, start = 1, sampling = 1, family = c("gauss", "gaussKern"),
  param = NULL)
```

Arguments

n	number of data points to simulate
values	a numeric vector specifying signal amplitudes for different states
rates	a square matrix matching the dimension of values each with rates[i,j] specifying the transition rate from state i to state j; the diagonal entries are ignored
start	the state in which the Markov chain is started
sampling	the sampling rate
family	whether Gaussian white ("gauss") or coloured ("gaussKern"), i.e. filtered, noise should be added; cf. family
param	for family="gauss", a single non-negative numeric specifying the standard deviation of the noise; for family="gaussKern", param must be a list with entry df giving the dfilter object used for filtering, an integer entry over which specifies the oversampling factor of the filter, i.e. param$df has to be created for a sampling rate of sampling times over, and an additional non-negative numeric entry sd specifying the noise's standard deviation *after* filtering; cf. family

Value

A list with components

cont	an object of class stepblock containing the simulated true values in continuous time, with an additional column state specifying the corresponding state
discr	an object of class stepblock containing the simulated true values reduced to discrete time, i.e. containing only the observable blocks
data	a data.frame with columns x and y containing the times and values of the simulated observations, respectively

Note

This follows the description for simulating ion channels given by VanDongen (1996).

References

VanDongen, A. M. J. (1996) A new algorithm for idealizing single ion channel data containing multiple unknown conductance levels. *Biophysical Journal* **70**(3), 1303–1315.

See Also

stepblock, jsmurf, stepbound, steppath, family, dfilter

Examples

```
# Simulate filtered ion channel recording with two states
set.seed(9)
# sampling rate 10 kHz
sampling <- 1e4
# tenfold oversampling
over <- 10
# 1 kHz 4-pole Bessel-filter, adjusted for oversampling
cutoff <- 1e3
df <- dfilter("bessel", list(pole=4, cutoff=cutoff / sampling / over))
# two states, leaving state 1 at 1 Hz, state 2 at 10 Hz
rates <- rbind(c(0, 1e0), c(1e1, 0))
# simulate 5 s, level 0 corresponds to state 1, level 1 to state 2
# noise level is 0.1 after filtering
sim <- contMC(5 * sampling, 0:1, rates, sampling=sampling, family="gaussKern",
  param = list(df=df, over=over, sd=0.1))
sim$cont
plot(sim$data, pch = ".")
lines(sim$discr, col = "red")
# noise level after filtering, estimated from first block
sd(sim$data$y[1:sim$discr$rightIndex[1]])
# show autocovariance in first block
acf(ts(sim$data$y[1:sim$discr$rightIndex[1]], freq=sampling), type = "cov")
# power spectrum in first block
s <- spec.pgram(ts(sim$data$y[1:sim$discr$rightIndex[1]], freq=sampling), spans=c(200,90))
# cutoff frequency is where power spectrum is halved
abline(v=cutoff, h=s$spec[1] / 2, lty = 2)
```

`critVal` *Critical values*

Description

Computes the vector of critical values or the global quantile. This function offers two ways of computation, either at significance level `alpha` from a Monte-Carlo simulation, see also section 3.2 in the vignette for more details, or from the global quantile / critical values given in the argument `q`. For more details on these two options see Section *Computation of critical values / global quantile*. Since a Monte-Carlo simulation lasts potentially much longer (up to several hours or days if the number of observations is in the millions) than the main calculations, this package saves them by default in the workspace and on the file system such that a second call requiring the same Monte-Carlo simulation will be much faster. For more details, in particular to which arguments the Monte-Carlo simulations are specific, see Section *Storing of Monte-Carlo simulations* below. Progress of a Monte-Carlo simulation can be reported by the argument `messages` in `...`, explained in `monteCarloSimulation`, and the saving can be controlled by the argument `option`.

Usage

```
critVal(n, q = NULL, alpha = NULL, nq = 2L^(as.integer(log2(n) + 1e-12) + 1L) - 1L,
        family = NULL, intervalSystem = NULL, lengths = NULL, penalty = NULL,
```

```
weights = NULL, stat = NULL, r = 1e4, output = c("vector", "value"),
options = NULL, ...)
```

Arguments

n	a positive integer giving the number of observations
q	either NULL, then the vector of critical values at level alpha will be computed from a Monte-Carlo simulation, or a numeric giving the global quantile or a numeric vector giving the vector of critical values. For more detailed information, in particular of which length the numeric vector should be, see Section *Computation of critical values / global quantile*. Either q or alpha must be given. Otherwise, alpha == 0.5 is chosen with a warning. Please note that by default the Monte-Carlo simulation will be saved in the workspace and on the file system, for more details see Section *Simulating, saving and loading of Monte-Carlo simulations* below
alpha	a probability, i.e. a single numeric between 0 and 1, giving the significance level. Its choice is a trade-off between data fit and parsimony of the estimator. In other words, this argument balances the risks of missing change-points and detecting additional artefacts. For more details on this choice see (Frick et al., 2014, section 4) and (Pein et al., 2017, section 3.4). Either q or alpha must be given. Otherwise, alpha == 0.5 is chosen with a warning
nq	a positive integer larger than or equal to n giving the (increased) number of observations for the Monte-Carlo simulation. See Section *Simulating, saving and loading of Monte-Carlo simulations* for more details
family	a string specifying the assumed parametric family, for more details see parametricFamily, currently "gauss", "hsmuce" and "mDependentPS" are supported. By default (NULL) "gauss" is assumed
intervalSystem	a string giving the used interval system, either "all" for all intervals, "dyaLen" for all intervals of dyadic length or "dyaPar" for the dyadic partition, for more details see intervalSystem. By default (NULL) the default interval system of the specified parametric family will be used, which one this will be is described in parametricFamily
lengths	an integer vector giving the set of lengths, i.e. only intervals of these lengths will be considered. Note that not all lengths are possible for all interval systems and for all parametric families, see intervalSystem and parametricFamily, respectively, to see which ones are allowed. By default (NULL) all lengths that are possible for the specified intervalSystem and for the specified parametric family will be used
penalty	a string specifying how different scales will be balanced, either "sqrt", "weights", "log" or "none", see penalty and section 3.2 in the vignette for more details. By default (NULL) the default penalty of the specified parametric family will be used, which one this will be is described in parametricFamily
weights	a numeric vector of length length(lengths) with only positive entries giving the weights that will be used for penalty "weights", see penalty and section 3.2.2 in the vignette for more details. By default (NULL) equal weights will be used, i.e.

weights == rep(1 / length(lengths), length(lengths))

stat — an object of class "MCSimulationVector" or "MCSimulationMaximum" giving a Monte-Carlo simulations, usually computed by monteCarloSimulation. If penalty == "weights" only "MCSimulationVector" is allowed. Has to be simulated for at least the given number of observations n and for the given family, intervalSystem and if "MCSimulationMaximum" for the given lengths and penalty. By default (NULL) the required simulation will be made available automatically accordingly to the given options. For more details see Section *Simulating, saving and loading of Monte-Carlo simulations* and section 3.4 in the vignette

r — a positive integer giving the required number of Monte-Carlo simulations if they will be simulated or loaded from the workspace or the file system

output — a string specifying the return value, if output == "vector" the vector of critical values will be computed and if output == "value" the global quantile will be computed. For penalty == "weights" the output must be "vector", since no global quantile can be determined for this penalty

options — a list specifying how Monte-Carlo simulations will be simulated, saved and loaded. For more details see Section *Simulating, saving and loading of Monte-Carlo simulations* and section 3.4 in the vignette

... — there are two groups of further arguments:

- further parameters of the parametric family. Depending on the argument family some might be required, but others might be optional, please see parametricFamily for more details
- further arguments (seed, rand.gen and messages) that will be passed to monteCarloSimulation. monteCarloSimulation will be called automatically and most of the arguments will be set accordingly to the arguments of critVal, no user interaction is required and possible for these parameters. In addition, seed, rand.gen and messages can be passed by the user

Value

If output == "vector" a numeric vector giving the vector of critical values, i.e. a vector of length length(lengths), giving for each length the corresponding critical value. If output == "value" a single numeric giving the global quantile. In both cases, additionally, an attribute "n" gives the number of observations for which the Monte-Carlo simulation was performed.

Computation of critical values / global quantile

This function offers two ways to compute the resulting value:

- If q == NULL it will be computed at significance level alpha from a Monte-Carlo simulation. For penalties "sqrt", "log" and "none" the global quantile will be the (1-alpha)-quantile of the penalised multiscale statistic, see section 3.2.1 in the vignette. And if required the vector of critical values will be derived from it. For penalty "weights" the vector of critical values will be calculated accordingly to the given weights. The Monte-Carlo simulation can either be given in stat or will be attempted to load or will be simulated. How Monte-Carlo simulations are simulated, saved and loaded can be controlled by the argument option, for more details see the Section *Simulating, saving and loading of Monte-Carlo simulations*.

- If q is given it will be derived from it. For the argument q either a single finite numeric giving the global quantile or a vector of finite numerics giving the vector of critical values (not allowed for output == "value") is possible:
  - A single numeric giving the global quantile. If output == "vector" the vector of critical values will be computed from it for the given lengths and penalty (penalty "weights" is not allowed). Note that the global quantile is specific to the arguments family, intervalSystem, lengths and penalty.
  - A vector of length length(lengths), giving for each length the corresponding critical value. This vector is identical to the vector of critical values.
  - A vector of length n giving for each length 1:n the corresponding critical value.
  - A vector of length equal to the number of all possible lengths for the given interval system and the given parametric family giving for each possible length the corresponding critical value.

  Additionally, an attribute "n" giving the number of observations for which q was computed is allowed. This attribute must be a single integer and equal to or larger than the argument n which means that q must have been computed for at least n observations. This allows additionally:
  - A vector of length attr(q, "n") giving for each length 1:attr(q, "n") the corresponding critical value.
  - A vector of length equal to the number of all possible lengths for the given interval system and the given parametric family if the number of observations is attr(q, "n") giving for each possible length the corresponding critical value.

  The attribute "n" will be kept or set to n if missing.

Simulating, saving and loading of Monte-Carlo simulations

Since a Monte-Carlo simulation lasts potentially much longer (up to several hours or days if the number of observations is in the millions) than the main calculations, this function offers multiple possibilities for saving and loading the simulations. The simulation, saving and loading can be controlled by the argument option. This argument has to be a list or NULL and the following named entries are allowed: "simulation", "save", "load", "envir" and "dirs". All missing entries will be set to their default option.
Objects of class "MCSimulationVector", containing simulations of the multiscale vector of statistics, and objects of class "MCSimulationMaximum", containing simulations of the penalised multiscale statistic (for penalties "sqrt", "log" and "none"), can be simulated, saved and loaded. Each Monte-Carlo simulation is specific to the number of observations, the parametric family and the interval system, for "MCSimulationMaximum" additionally to the set of lengths and the used penalty. Both types will lead to the same result, however, an object of class "MCSimulationVector" is more flexible, since critical values for all penalties and all set of lengths can be derived from it, but requires much more storage space and has slightly larger saving and loading times. Note that Monte-Carlo simulations can only be saved and loaded if they are generated with the default function for generating random observations, i.e. when rand.gen (in ...) is NULL. For a given simulation this is signalled by the attribute "save" which is TRUE if a simulation can be saved and FALSE otherwise.
Monte-Carlo simulations can also be performed for a (slightly) larger number of observations n_q given in the argument nq, which avoids extensive resimulations for only a little bit varying number of observations. The overestimation control is still satisfied but the detection power is (slightly)

smaller. But note that the default lengths might change when the number of observations is increased and, hence, for type "vectorIncreased" still a different simulation might be required.
We refer to the different types as follow:

- "vector": an object of class "MCSimulationMaximum", i.e. simulations of the penalized multiscale statistic, for n observations
- "vectorIncreased": an object of class "MCSimulationMaximum", i.e. simulations of the penalized multiscale statistic, for nq observations
- "matrix": an object of class "MCSimulationVector", i.e. simulations of the multiscale vector of statistics, for n observations
- "matrixIncreased": an object of class "MCSimulationVector", i.e. simulations of the multiscale vector of statistics, for nq observations

The simulations can either be saved in the workspace in the variable critValStepRTab or persistently on the file system for which the package R.cache is used. Loading from the workspace is faster, but either the user has to store the workspace manually or in a new session simulations have to be performed again. Moreover, storing in and loading from variables and RDS files is supported. Finally, a pre-computed collection of simulations of type "matrixIncreased" for parametric families "gauss" and "hsmuce" can be accessed by installing the package stepRdata available from http://www.stochastik.math.uni-goettingen.de/stepRdata_1.0-0.tar.gz.

options$envir and options$dirs: For loading from / saving in the workspace the variable critValStepRTab in the environment options$envir will be looked for and if missing in case of saving also created there. Moreover, the variable(s) specified in options$save$variable (explained in the Subsection *Saving: options$save*) will be assigned to this environment. options$envir will be passed to the arguments pos and where in the functions assign, get, and exists, respectively. By default, a local enviroment in the package is used.
For loading from / saving on the file system loadCache(key = keyList, dirs = options$dirs) and saveCache(stat, key = attr(stat, "keyList"), dirs = options$dirs) are called, respectively. In other words, options$dirs has to be a character vector constituting the path to the cache subdirectory relative to the cache root directory as returned by getCacheRootPath(). If options$dirs == "" the path will be the cache root path. By default the subdirectory "stepR" is used, i.e. options$dirs == "stepR". Missing directories will be created.

Simulation: options$simulation: Whenever Monte-Carlo simulations have to be performed, i.e. when stat == NULL and the required Monte-Carlo simulation could not be loaded, the type specified in options$simulation will be simulated by monteCarloSimulation. In other words, options$simulation must be a single string of the following: "vector", "vectorIncreased", "matrix" or "matrixIncreased". By default (options$simulation == NULL), an object of class "MCSimulationVector" for nq observations will be simulated, i.e. options$simulation == "matrixIncreased". For this choice please recall the explanations regarding computation time and flexibility at the beginning of this section.

Loading: options$load: Loading of the simulations can be controlled by the entry options$load which itself has to be a list with possible entries: "RDSfile", "workspace", "package" and "fileSystem". Missing entries disable the loading from this option. Whenever a Monte-Carlo simulation is required, i.e. when the variable q is not given, it will be searched for at the following places in the given order until found:

1. in the variable `stat`,
2. in `options$load$RDSfile` as an RDS file, i.e. the simulation will be loaded by
   `readRDS(options$load$RDSfile)`.
   In other words, `options$load$RDSfile` has to be a `connection` or the name of the file where the R object is read from,
3. in the workspace or on the file system in the following order: `"vector"`, `"matrix"`, `"vectorIncreased"` and finally of `"matrixIncreased"`. For `penalty == "weights"` it will only be looked for `"matrix"` and `"matrixIncreased"`. For each options it will first be looked in the workspace and then on the file system. All searches can be disabled by not specifying the corresponding string in `options$load$workspace` and `options$load$fileSystem`. In other words, `options$load$workspace` and `options$load$fileSystem` have to be vectors of strings containing none, some or all of `"vector"`, `"matrix"`, `"vectorIncreased"` and `"matrixIncreased"`,
4. in the package `stepRdata` (if installed) and if `options$load$package == TRUE`. In other words, `options$load$package` must be a single logical or `NULL`,
5. if all other options fail a Monte-Carlo simulation will be performed.

By default (if `options$load` is missing / `NULL`) no RDS file is specified and all other options are enabled, i.e.

```
options$load <- list(workspace = c("vector", "vectorIncreased",
                                   "matrix", "matrixIncreased"),
                     fileSystem = c("vector", "vectorIncreased",
                                    "matrix", "matrixIncreased"),
                     package = TRUE, RDSfile = NULL).
```

Saving: options$save: Saving of the simulations can be controlled by the entry `options$save` which itself has to be a `list` with possible entries: `"workspace"`, `"fileSystem"`, `"RDSfile"` and `"variable"`. Missing entries disable the saving in this option.
All available simulations, no matter whether they are given by `stat`, loaded, simulated or in case of `"vector"` and `"vectorIncreased"` computed from `"matrix"` and `"matrixIncreased"`, respectively, will be saved in all options for which the corresponding type is specified. Here we say a simulation is of type `"vectorIncreased"` or `"matrixIncreased"` if the simulation is not performed for `n` observations. More specifically, a simulation will be saved:

1. in the workspace or on the file system if the corresponding string is contained in `options$save$workspace` and `options$save$fileSystem`, respectively. In other words, `options$save$workspace` and `options$save$fileSystem` have to be vectors of strings containing none, some or all of `"vector"`, `"matrix"`, `"vectorIncreased"` and `"matrixIncreased"`,
2. in an RDS file specified by `options$save$RDSfile` which has to be a vector of one or two `connections` or names of files where the R object is saved to. If `options$save$RDSfile` is of length two a simulation of type `"vector"` or `"vectorIncreased"` (only one can occur at one function call) will be saved in `options$save$RDSfile[1]` by
   `saveRDS(stat, file = options$save$RDSfile[1])`
   and `"matrix"` or `"matrixIncreased"` (only one can occur at one function call) will be saved in `options$save$RDSfile[2]`. If `options$save$RDSfile` is of length one both will be saved in `options$save$RDSfile` which means if both occur at the same call only `"vector"` or `"vectorIncreased"` will be saved. Each saving can be disabled by not specifying `options$save$RDSfile` or by passing an empty string to the corresponding entry of `options$save$RDSfile`.

3. in a variable named by `options$save$variable` in the `environment` `options$envir`. Hence, `options$save$variable` has to be a vector of one or two containing variable names (character vectors). If `options$save$variable` is of length two a simulation of type `"vector"` or `"vectorIncreased"` (only one can occur at one function call) will be saved in `options$save$variable[1]` and `"matrix"` or `"matrixIncreased"` (only one can occur at one function call) will be saved in `options$save$variable[2]`. If `options$save$variable` is of length one both will be saved in `options$save$variable` which means if both occur at the same call only `"vector"` or `"vectorIncreased"` will be saved. Each saving can be disabled by not specifying `options$save$variable` or by passing `""` to the corresponding entry of `options$save$variable`.

By default (if `options$save` is missing) `"vector"` and `"vectorIncreased"` will be saved in the workspace and `"matrix"` and `"matrixIncreased"` on the file system, i.e.

```
options$save <- list(workspace = c("vector", "vectorIncreased"),
                     fileSystem = c("matrix", "matrixIncreased"),
                     RDSfile = NULL, variable = NULL).
```

Simulations can be removed from the workspace by removing the variable `critValStepRTab`, i.e. by calling `remove(critValStepRTab, envir = envir)`, with `envir` the used environment, and from the file system by deleting the corresponding subfolder, i.e. by calling

```
unlink(file.path(R.cache::getCacheRootPath(), dirs), recursive = TRUE),
```

with `dirs` the corresponding subdirectory.

References

Frick, K., Munk, A., Sieling, H. (2014) Multiscale change-point inference. With discussion and rejoinder by the authors. *Journal of the Royal Statistical Society, Series B* **76**(3), 495–580.

Pein, F., Sieling, H., Munk, A. (2017) Heterogeneous change point inference. *Journal of the Royal Statistical Society, Series B*, **79**(4), 1207–1227.

See Also

`monteCarloSimulation`, `penalty`, `parametricFamily`, `intervalSystem`, `stepFit`, `computeBounds`

Examples

```
# vector of critical values
qVector <- critVal(100L, alpha = 0.5)
# global quantile
qValue <- critVal(100L, alpha = 0.5, output = "value")

# vector can be computed from the global quantile
identical(critVal(100L, q = qValue), qVector)

# for a conservative significance level, stronger confidence statements
critVal(100L, alpha = 0.05)
critVal(100L, alpha = 0.05, output = "value")

# higher significance level for larger detection power, but less confidence
critVal(100L, alpha = 0.99)
```

```
critVal(100L, alpha = 0.99, output = "value")

# different parametric family, different intervalSystem, a subset of lengths,
# different penalty and given weights
q <- critVal(100L, alpha = 0.05, family = "hsmuce", intervalSystem = "dyaLen",
             lengths = c(2L, 4L, 16L, 32L), penalty = "weights",
             weights = c(0.4, 0.3, 0.2, 0.1))

# vector of critical values can be given by a vector of length n
vec <- 1:100
vec[c(2L, 4L, 16L, 32L)] <- q
attr(vec, "n") <- 128L
identical(critVal(100L, q = vec, family = "hsmuce", intervalSystem = "dyaLen",
                  lengths = c(2L, 4L, 16L, 32L)), q)

# with a given monte-Carlo simulation for nq = 128 observations
stat <- monteCarloSimulation(128)
critVal(n = 100L, alpha = 0.05, stat = stat)

# the above calls saved and (attempted to) load Monte-Carlo simulations and
# simulated them for nq = 128 observations
# in the following call no saving, no loading and simulation for n = 100
# observations is required, progress of the simulation will be reported
critVal(n = 100L, alpha = 0.05, messages = 1000L,
        options = list(simulation = "vector", load = list(), save = list()))

# only type "vector" will be saved and loaded in the workspace
critVal(n = 100L, alpha = 0.05, messages = 1000L,
        options = list(simulation = "vector", load = list(workspace = "vector"),
                       save = list(workspace = "vector")))

# simulation of type "matrix" will be saved in a RDS file
# saving of type "vector" is disabled by passing "",
# different seed is set and number of simulations is reduced to r = 1e3
# to allow faster computation at the price of a less precise result
file <- tempfile(pattern = "file", tmpdir = tempdir(), fileext = ".RDS")
critVal(n = 100L, alpha = 0.05, seed = 1, r = 1e3,
        options = list(simulation = "matrix", load = list(),
                       save = list(RDSfile = c("", file))))
identical(readRDS(file), monteCarloSimulation(100L, seed = 1, r = 1e3))
```

`dfilter` *Digital filters*

Description

Create digital filters.

Deprecation warning: This function is mainly used for patchlamp recordings and may be transferred to a specialised package.

Usage

```
dfilter(type = c("bessel", "gauss", "custom"), param = list(pole = 4, cutoff = 1 / 10),
  len = ceiling(3/param$cutoff))
## S3 method for class 'dfilter'
print(x, ...)
```

Arguments

type	allows to choose Bessel, Gauss or custom filters
param	for a "bessel" filter a list with entries pole and cutoff giving the filter's number of poles (order) and cut-off frequency, resp.; for a "gauss" filter the filter's bandwidth (standard deviation) as a single numeric; for a custom filter either a numeric vector specifying the filter's kernel or a list with items kern and step of the same length giving the filter's kernel and step-response, resp.
len	filter length (unnecessary for "custom" filters
x	the object
...	for generic methods only

Value

Returns a list of class dfilter that contains elements kern and step, the (digitised) filter kernel and step-response, resp., as well as an element param containing the argument param, for a "bessel" filter alongside the corresponding analogue kernel, step response, power spectrum, and autocorrelation function depending on time or frequency as elements kernfun, stepfun, spectrum, and acfun, resp.

See Also

filter, convolve, BesselPolynomial, Normal, family

Examples

```
# 6-pole Bessel filter with cut-off frequency 1 / 100, with length 100 (too short!)
dfilter("bessel", list(pole = 6, cutoff = 1 / 100), 100)
# custom filter: running mean of length 3
dfilter("custom", rep(1, 3))
dfilter("custom", rep(1, 3))$kern # normalised!
dfilter("custom", rep(1, 3))$step
# Gaussian filter with bandwidth 3 and length 11 (from -5 to 5)
dfilter("gauss", 3, 11)
```

family *Family of distributions*

Description

Families of distributions supported by package stepR.

Deprecation warning: This overviw is deprecated, but still given and up to date for some older, deprecated functions, however, may be removed in a future version. For an overview about the parametric families supported by the new functions see parametricFamily.

Details

Package stepR supports several families of distributions (mainly exponential) to model the data, some of which require additional (fixed) parameters. In particular, the following families are available:

"gauss" normal distribution with unknown mean but known, fixed standard deviation given as a single numeric (will be estimated using sdrobnorm if omitted); cf. dnorm.

"gaussvar" normal distribution with unknown variance but known, fixed mean assumed to be zero; cf. dnorm.

"poisson" Poisson distribution with unknown intensity (no additional parameter); cf. dpois.

"binomial" binomial distribution with unknown success probability but known, fixed size given as a single integer; cf. dbinom.

"gaussKern" normal distribution with unknown mean and unknown, fixed standard deviation (being estimated using sdrobnorm), after filtering with a fixed filter which needs to be given as the additional parameter (a dfilter object); cf. dfilter.

The family is selected via the family argument, providing the corresponding string, while the param argument contains the parameters if any.

Note

Beware that not all families can be chosen for all functions.

See Also

Distributions, parametricFamily, dnorm, dpois, dbinom, dfilter, sdrobnorm

Examples

```
# illustrating different families fitted to the same binomial data set
size <- 200
n <- 200
# truth
p <- 10^seq(-3, -0.1, length = n)
# data
y <- rbinom(n, size, p)
```

```
plot(y)
lines(size * p, col = "red")
# fit 4 jumps, binomial family
jumps <- 4
bfit <- steppath(y, family = "binomial", param = size, max.blocks = jumps)
lines(bfit[[jumps]], col = "orange")
# Gaussian approximation with estimated variance
gfit <- steppath(y, family = "gauss", max.blocks = jumps)
lines(gfit[[jumps]], col = "green3", lty = 2)
# Poisson approximation
pfit <- steppath(y, family = "poisson", max.blocks = jumps)
lines(pfit[[jumps]], col = "blue", lty = 2)
legend("topleft", legend = c("binomial", "gauss", "poisson"), lwd = 2,
  col = c("orange", "green3", "blue"))
```

`intervalSystem` *Interval systems*

Description

Overview about the supported interval systems. More details are given in section 6 of the vignette.

Details

The following interval systems (set of intervals on which tests will be performed) are available. Intervals are given as indices of observations / sample points.

`"all"` all intervals. More precisely, the set $\{[i, j], 1 \leq i \leq j \leq n\}$. This system allows all lengths `1:n`.

`"dyaLen"` all intervals of dyadic length. More precisely, the set $\{[i, j], 1 \leq i \leq j \leq n \; s.t. \; j - i + 1 = 2^k, \; k \in N_0\}$. This system allows all lengths of dyadic length `2^(0:as.integer(floor(log2(n)) + 1e-6))`.

`"dyaPar"` the dyadic partition, i.e. all disjoint intervals of dyadic length. More precisely, the set $\{[(i-1) * 2^k + 1, i * 2^k], i = 1, \ldots, \lfloor n/2^k \rfloor, \; k = 0, \ldots, \lfloor \log_2(n) \rfloor\}$. This system allows all lengths of dyadic length `2^(0:as.integer(floor(log2(n)) + 1e-6))`.

The interval system is selected via the `intervalSystem` argument, providing the corresponding string. By default (`NULL`) the default interval system of the specified parametric family will be used, which one this will be is described in `parametricFamily`. With the additional argument `lengths` it is possible to specify a set of lengths such that only tests on intervals with a length contained in this set will be performed. The set of lengths has to be a subset of all lengths that are allowed by the interval system and the parametric family. By default (`NULL`) all lengths allowed by the interval system and the parametric family are used.

See Also

parametricFamily

Examples

```
y <- c(rnorm(50), rnorm(50, 2))

# interval system of all intervals and all lengths
fit <- stepFit(y, alpha = 0.5, intervalSystem = "all", lengths = 1:100,
               jumpint = TRUE, confband = TRUE)

# default for family "gauss" if number of observations is 1000 or less
identical(stepFit(y, alpha = 0.5, jumpint = TRUE, confband = TRUE), fit)

# intervalSystem "dyaLen" and a subset of lengths
!identical(stepFit(y, alpha = 0.5, intervalSystem = "dyaLen", lengths = c(2, 4, 16),
                   jumpint = TRUE, confband = TRUE), fit)

# default for lengths are all possible lengths of the interval system
# and the parametric family
identical(stepFit(y, alpha = 0.5, intervalSystem = "dyaPar",
                  jumpint = TRUE, confband = TRUE),
          stepFit(y, alpha = 0.5, intervalSystem = "dyaPar", lengths = 2^(0:6),
                  jumpint = TRUE, confband = TRUE))

# interval system "dyaPar" is default for parametric family "hsmuce"
# length 1 is not possible for this parametric family
identical(stepFit(y, alpha = 0.5, family = "hsmuce",
                  jumpint = TRUE, confband = TRUE),
          stepFit(y, alpha = 0.5, family = "hsmuce", intervalSystem = "dyaPar",
                  lengths = 2^(1:6), jumpint = TRUE, confband = TRUE))

# interval system "dyaLen" is default for parametric family "mDependentPS"
identical(stepFit(y, alpha = 0.5, family = "mDependentPS", covariances = c(1, 0.5),
                  jumpint = TRUE, confband = TRUE),
          stepFit(y, alpha = 0.5, family = "mDependentPS", covariances = c(1, 0.5),
                  intervalSystem = "dyaLen", lengths = 2^(0:6),
                  jumpint = TRUE, confband = TRUE))
```

`jsmurf` *Reconstruct filtered piecewise constant functions with noise*

Description

Reconstructs a piecewise constant function to which white noise was added and the sum filtered afterwards.

Deprecation warning: This function is mainly used for patchlamp recordings and may be transferred to a specialised package.

Usage

```
jsmurf(y, x = 1:length(y), x0 = 2 * x[1] - x[2], q, alpha = 0.05, r = 4e3,
  lengths = 2^(floor(log2(length(y))):floor(log2(max(length(param$kern) + 1,
    1 / param$param$cutoff)))), param, rm.out = FALSE,
    jumpint = confband, confband = FALSE)
```

Arguments

y	a numeric vector containing the serial data
x	a numeric vector of the same length as y containing the corresponding sample points
x0	a single numeric giving the last unobserved sample point directly before sampling started
q	threshold value, by default chosen automatically
alpha	significance level; if set to a value in (0,1), q is chosen as the corresponding quantile of the asymptotic (if r is not given) null distribution (and any value specified for q is silently ignored)
r	numer of simulations; if specified along alpha, q is chosen as the corresponding quantile of the simulated null distribution
lengths	length of intervals considered; by default up to a sample size of 1000 all lengths, otherwise only dyadic lengths
param	a dfilter object specifiying the filter
rm.out	a logical specifying whether outliers should be removed prior to the analysis
jumpint	logical (FALSE by default), indicates if confidence sets for jumps should be computed
confband	logical, indicates if a confidence band for the piecewise-continuous function should be computed

Value

An object object of class stepfit that contains the fit; if jumpint == TRUE function jumpint allows to extract the 1 - alpha confidence interval for the jumps, if confband == TRUE function confband allows to extract the 1 - alpha confidence band.

References

Hotz, T., Schütte, O., Sieling, H., Polupanow, T., Diederichsen, U., Steinem, C., and Munk, A. (2013) Idealizing ion channel recordings by a jump segmentation multiresolution filter. *IEEE Transactions on NanoBioscience* **12**(4), 376–386.

See Also

stepbound, bounds, family, MRC.asymptotic, sdrobnorm, stepfit

Examples

```
# simulate filtered ion channel recording with two states
set.seed(9)
# sampling rate 10 kHz
sampling <- 1e4
# tenfold oversampling
over <- 10
# 1 kHz 4-pole Bessel-filter, adjusted for oversampling
cutoff <- 1e3
df.over <- dfilter("bessel", list(pole=4, cutoff=cutoff / sampling / over))
# two states, leaving state 1 at 10 Hz, state 2 at 20 Hz
rates <- rbind(c(0, 10), c(20, 0))
# simulate 0.5 s, level 0 corresponds to state 1, level 1 to state 2
# noise level is 0.3 after filtering
sim <- contMC(0.5 * sampling, 0:1, rates, sampling=sampling, family="gaussKern",
  param = list(df=df.over, over=over, sd=0.3))
plot(sim$data, pch = ".")
lines(sim$discr, col = "red")
# fit using filter corresponding to sample rate
df <- dfilter("bessel", list(pole=4, cutoff=cutoff / sampling))
fit <- jsmurf(sim$data$y, sim$data$x, param=df, r=1e2)
lines(fit, col = "blue")
# fitted values take filter into account
lines(sim$data$x, fitted(fit), col = "green3", lty = 2)
```

---

jumpint *Confidence intervals for jumps and confidence bands for step functions*

Description

Extract and plot confidence intervals and bands from fits given by a `stepfit` object.

Usage

```
jumpint(sb, ...)
## S3 method for class 'stepfit'
jumpint(sb, ...)
## S3 method for class 'jumpint'
points(x, pch.left = NA, pch.right = NA, y.left = NA, y.right = NA, xpd = NA, ...)
confband(sb, ...)
## S3 method for class 'stepfit'
confband(sb, ...)
## S3 method for class 'confband'
lines(x, dataspace = TRUE, ...)
```

Arguments

sb	the result of a fit by `stepbound`

x	the object
pch.left, pch.right	the plotting character to use for the left/right end of the interval with defaults "(" and "]" (see parameter pch of par)
y.left, y.right	at which height to plot the interval boundaries with default par()$usr[3]
xpd	see par
dataspace	logical determining whether the expected value should be plotted instead of the fitted parameter value, useful e.g. for family = "binomial", where it will plot the fitted success probability times the number of trials per observation
...	arguments to be passed to generic methods

Value

For jumpint an object of class jumpint, i.e. a data.frame whose columns rightEndLeftBound and rightEndRightBound specify the left and right end of the confidence interval for the block's right end, resp., given the number of blocks was estimated correctly, and similarly columns rightIndexLeftBound and rightIndexRightBound specify the left and right indices of the confidence interval, resp. Function points plots these intervals on the lower horizontal axis (by default).

For confband an object of class confband, i.e. a data.frame with columns lower and upper specifying a confidence band computed at every point x; this is a simultaneous confidence band assuming the true number of jumps has been determined. Function lines plots the confidence band.

Note

Observe that jumps may occur immediately before or after an observed x; this lack of knowledge is reflected in the visual impressions by the lower and upper envelopes jumping vertically early, so that possible jumps between xs remain within the band, and by the confidence intervals starting immediately after the last x for which there cannot be a jump, cf. the note in the help for stepblock.

See Also

stepbound, points, lines

Examples

```
# simulate Bernoulli data with four blocks
y <- rbinom(200, 1, rep(c(0.1, 0.7, 0.3, 0.9), each=50))
# fit step function
sb <- stepbound(y, family="binomial", param=1, confband=TRUE)
plot(y, pch="|")
lines(sb)
# confidence intervals for jumps
jumpint(sb)
points(jumpint(sb), col="blue")
# confidence band
confband(sb)
lines(confband(sb), lty=2, col="blue")
```

`monteCarloSimulation` *Monte Carlo simulation*

Description

Performs Monte-Carlo simulations of the multiscale vector of statistics, (3.9) in the vignette, and of the penalised multiscale statistic, (3.6) in the vignette, when no signal is present, see also section 3.2.3 in the vignette.

Usage

```
monteCarloSimulation(n, r = 1e4L, family = NULL, intervalSystem = NULL,
                     lengths = NULL, penalty = NULL,
                     output = c("vector", "maximum"), seed = n,
                     rand.gen = NULL, messages = NULL, ...)
```

Arguments

`n`	a positive integer giving the number of observations for which the Monte-Carlo simulation will be performed
`r`	a positive integer giving the number of repititions
`family`	a string specifying the assumed parametric family, for more details see parametricFamily, currently `"gauss"`, `"hsmuce"` and `"mDependentPS"` are supported. By default (`NULL`) `"gauss"` is assumed
`intervalSystem`	a string giving the used interval system, either `"all"` for all intervals, `"dyaLen"` for all intervals of dyadic length or `"dyaPar"` for the dyadic partition, for more details see intervalSystem. By default (`NULL`) the default interval system of the specified parametric family will be used, which one this will be is described in `parametricFamily`
`lengths`	an integer vector giving the set of lengths, i.e. only intervals of these lengths will be considered. Only required for `output == "maximum"`, otherwise ignored with a `warning`. Note that not all lengths are possible for all interval systems and for all parametric families, see intervalSystem and parametricFamily, respectively, to see which ones are allowed. By default (`NULL`) all lengths that are possible for the specified `intervalSystem` and for the specified parametric family will be used
`penalty`	a string specifying how the statistics will be penalised, either `"sqrt"`, `"log"` or `"none"`, see penalty and section 3.2 in the vignette for more details. Only required for `output == "maximum"`, otherwise ignored with a `warning`. By default (`NULL`) the default penalty of the specified parametric family will be used, which one this will be is described in `parametricFamily`
`output`	a string specifying the output, see *Value*
`seed`	will be passed to `set.seed` to set a seed, `set.seed` will not be called if this argument is set to `"no"`, i.e. a single value, interpreted as an `integer`, `NULL` or `"no"`

rand.gen	by default (NULL) this argument will be replaced by the default function to generate random observations of the given family. Note that a Monte-Carlo simulation can only be saved if rand.gen == NULL. Alternatively, an own function expecting a single argument named data and returning a numeric vector of length n, this is given by data$n. Will be called with rand.gen(data = data), with data a list containing the named entries n, the expected number of data points, and parameters of the parametric family, e.g. sd for family == "gauss" or covariances for family == "mDependentPS"
messages	a positive integer or NULL, in each messages iteration a message will be printed in order to show the progress of the simulation, if NULL no message will be given
...	further parameters of the parametric family. Depending on the argument family some might be required, but others might be optional, please see parametricFamily for more details

Value

If output == "vector" an object of class "MCSimulationVector", i.e. a d_n times r matrix containing r independent samples of the multiscale vector of statistics, with d_n the number of scales, i.e. the number of possible lengths for the given interval system and given parametric family. If output == "maximum" an object of class "MCSimulationMaximum", i.e. a vector of length r containing r independent samples of the penalised multiscale statistic. For both, additionally, the following attributes are set:

- "keyList": A list specifying for which number of observations n, which parametric family with which parameters by a SHA-1 hash, which interval system and in case of "MCSimulationMaximum", additionally, for which lengths and which penalisation the simulation was performed.
- "key": A key used internally for identification when the object will be saved and loaded.
- "n": The number of observations n for which the simulation was performed.
- "lengths": The lengths for which the simulation was performed.
- "save": A logical which is TRUE if the object can be saved which is the case for rand.gen == NULL and FALSE otherwise.

References

Frick, K., Munk, A., Sieling, H. (2014) Multiscale change-point inference. With discussion and rejoinder by the authors. *Journal of the Royal Statistical Society, Series B* **76**(3), 495–580.

Pein, F., Sieling, H., Munk, A. (2017) Heterogeneous change point inference. *Journal of the Royal Statistical Society, Series B*, **79**(4), 1207–1227.

See Also

critVal, computeStat, penalty, parametricFamily, intervalSystem

Examples

```
# monteCarloSimulation will be called in critVal, can be called explicitly
# object of class MCSimulationVector
```

```
stat <- monteCarloSimulation(n = 100L)

identical(critVal(n = 100L, alpha = 0.5, stat = stat),
          critVal(n = 100L, alpha = 0.5,
                  options = list(load = list(), simulation = "matrix")))

# object of class MCSimulationMaximum
stat <- monteCarloSimulation(n = 100L, output = "maximum")
identical(critVal(n = 100L, alpha = 0.5, stat = stat),
          critVal(n = 100L, alpha = 0.5,
                  options = list(load = list(), simulation = "vector")))

# different interval system, lengths and penalty
monteCarloSimulation(n = 100L, output = "maximum", intervalSystem = "dyaLen",
              lengths = c(1L, 2L, 4L, 8L), penalty = "log")

# with a different number of iterations, different seed,
# reported progress and user written rand.gen function
stat <- monteCarloSimulation(n = 100L, r = 1e3, seed = 1, messages = 100,
                             rand.gen = function(data) {rnorm(100)})

# the optional argument sd of parametric family "gauss" will be replaced by 1
identical(monteCarloSimulation(n = 100L, r = 1e3, sd = 5),
          monteCarloSimulation(n = 100L, r = 1e3, sd = 1))

# simulation for family "hsmuce"
monteCarloSimulation(n = 100L, family = "hsmuce")

# simulation for family "mDependentGauss"
# covariances must be given (can also be given by correlations or filter)
stat <- monteCarloSimulation(n = 100L, family = "mDependentPS",
                             covariances = c(1, 0.5, 0.3))

# variance will be standardized to 1
# output might be on some systems even identical
all.equal(monteCarloSimulation(n = 100L, family = "mDependentPS",
                               covariances = c(2, 1, 0.6)), stat)
```

MRC *Compute Multiresolution Criterion*

Description

Computes multiresolution coefficients, the corresponding criterion, simulates these for Gaussian white or coloured noise, based on which p-values and quantiles are obtained.

Deprecation warning: The function `MRC.simul` is deprecated, but still working, however, may be defunct in a future version. Please use instead the function `monteCarloSimulation`. An example how to reproduce results is given below. Some other functions are help function and might be removed, too.

Usage

```
MRC(x, lengths = 2^(floor(log2(length(x))):0), norm = sqrt(lengths),
  penalty = c("none", "log", "sqrt"))
MRCoeff(x, lengths = 2^(floor(log2(length(x))):0), norm = sqrt(lengths), signed = FALSE)
MRC.simul(n, r, lengths = 2^(floor(log2(n)):0), penalty = c("none", "log", "sqrt"))
MRC.pvalue(q, n, r, lengths = 2^(floor(log2(n)):0), penalty = c("none", "log", "sqrt"),
  name = ".MRC.table", pos = .MCstepR, inherits = TRUE)
MRC.FFT(epsFFT, testFFT, K = matrix(TRUE, nrow(testFFT), ncol(testFFT)), lengths,
  penalty = c("none", "log", "sqrt"))
MRC.quant(p, n, r, lengths = 2^(floor(log2(n)):0), penalty = c("none", "log", "sqrt"),
  name = ".MRC.table", pos = .MCstepR, inherits = TRUE, ...)
kMRC.simul(n, r, kern, lengths = 2^(floor(log2(n)):ceiling(log2(length(kern)))))
kMRC.pvalue(q, n, r, kern, lengths = 2^(floor(log2(n)):ceiling(log2(length(kern)))),
  name = ".MRC.ktable", pos = .MCstepR, inherits = TRUE)
kMRC.quant(p, n, r, kern, lengths = 2^(floor(log2(n)):ceiling(log2(length(kern)))),
  name = ".MRC.ktable", pos = .MCstepR, inherits = TRUE, ...)
```

Arguments

`x`	a vector of numerical observations
`lengths`	vector of interval lengths to use, dyadic intervals by default
`signed`	whether signed coefficients should be returned
`q`	quantile
`n`	length of data set
`r`	number of simulations to use
`name`, `pos`, `inherits`	under which name and where precomputed results are stored, or retrieved, see `assign`
`K`	a `logical` matrix indicating the set of valid intervals
`epsFFT`	a vector containg the FFT of the data set
`testFFT`	a matrix containing the FFTs of the intervals
`kern`	a filter kernel
`penalty`	penalty term in the multiresolution statistic: `"none"` for no penalty, `"log"` for penalizing the log-length of an interval, and `"sqrt"` for penalizing the square root of the MRC; or a `function` taking two arguments, the first being the multiresolution coefficients, the second the interval lenghts
`norm`	how the partial sums should be normalised, by default `sqrt(lengths)`, so they are normalised to equal variance across all interval lengths
`p`	p-value
`...`	further arguments passed to function `quantile`

Value

`MRC`	a vector giving the maximum as well as the indices of the corresponding interval's start and length
`MRCoeff`	a matrix giving the multiresolution coefficients for all test intervals
`MRC.pvalue`, `MRC.quant`, `MRC.simul`	the corresponding p-value / quantile / vector of simulated values under the assumption of standard Gaussian white noise
`kMRC.pvalue`, `kMRC.simul`, `kMRC.simul`	the corresponding p-value / quantile / vector of simulated values under the assumption of filtered Gaussian white noise

References

Davies, P. L., Kovac, A. (2001) Local extremes, runs, strings and multiresolution. *The Annals of Statistics* **29**, 1–65.

Dümbgen, L., Spokoiny, V. (2001) Multiscale testing of qualitative hypotheses. *The Annals of Statistics* **29**, 124–152.

Siegmund, D. O., Venkatraman, E. S. (1995) Using the generalized likelihood ratio statistic for sequential detection of a change-point. *The Annals of Statistics* **23**, 255–271.

Siegmund, D. O., Yakir, B. (2000) Tail probabilities for the null distribution of scanning statistics. *Bernoulli* **6**, 191–213.

See Also

`monteCarloSimulation`, `smuceR`, `jsmurf`, `stepbound`, `stepsel`, `quantile`

Examples

```
set.seed(100)
all.equal(MRC.simul(100, r = 100),
          sort(monteCarloSimulation(n = 100, r = 100, output = "maximum",
                                    penalty = "none", intervalSystem = "dyaLen")),
          check.attributes = FALSE)

# simulate signal of 100 data points
set.seed(100)
f <- rep(c(0, 2, 0), c(60, 10, 30))
# add gaussian noise
x <- f + rnorm(100)
# compute multiresolution criterion
m <- MRC(x)
# compute Monte-Carlo p-value based on 100 simulations
MRC.pvalue(m["max"], length(x), 100)
# compute multiresolution coefficients
M <- MRCoeff(x)

# plot multiresolution coefficients, colours show p-values below 5% in 1% steps
op <- par(mar = c(5, 4, 2, 4) + 0.1)
image(1:length(x), seq(min(x), max(x), length = ncol(M)), apply(M[,ncol(M):1], 1:2,
```

```
  MRC.pvalue, n = length(x), r = 100), breaks = (0:5) / 100,
  col = rgb(1, seq(0, 1, length = 5), 0, 0.75),
  xlab = "location / left end of interval", ylab ="measurement",
  main = "Multiresolution Coefficients",
  sub = paste("MRC p-value =", signif(MRC.pvalue(m["max"], length(x), 100), 3)))
axis(4, min(x) + diff(range(x)) * ( pretty(1:ncol(M) - 1) ) / dim(M)[2],
  2^pretty(1:ncol(M) - 1))
mtext("interval lengths", 4, 3)
# plot signal and its mean
points(x)
lines(f, lty = 2)
abline(h = mean(x))
par(op)
```

`MRC.1000` *Values of the MRC statistic for 1,000 observations (all intervals)*

Description

Simulated values of the MRC statistic with `penalty="sqrt"` based on all interval lengths computed from Gaussian white noise sequences of length 1,000.

Deprecation warning: This data set is needed for `smuceR` and may be removed when this function will be removed.

Usage

```
MRC.1000
```

Format

A `numeric` vector containing 10,000 sorted values.

Examples

```
# threshold value for 95% confidence
quantile(stepR::MRC.1000, .95)
```

MRC.asymptotic *"Asymptotic" values of the MRC statistic (all intervals)*

Description

Simulated values of the MRC statistic with `penalty="sqrt"` based on all interval lengths computed from Gaussian white noise sequences of ("almost infinite") length 5,000.

Deprecation warning: This data set is needed for `smuceR` and may be removed when this function will be removed.

Usage

```
MRC.asymptotic
```

Format

A `numeric` vector containing 10,000 sorted values.

Examples

```
# "asymptotic" threshold value for 95% confidence
quantile(stepR::MRC.asymptotic, .95)
```

MRC.asymptotic.dyadic *"Asymptotic" values of the MRC statistic (dyadic intervals)*

Description

Simulated values of the MRC statistic with `penalty="sqrt"` based on dyadic interval lengths computed from Gaussian white noise sequences of ("almost infinite") length 100,000.

Deprecation warning: This data set is needed for `smuceR` and may be removed when this function will be removed.

Usage

```
MRC.asymptotic.dyadic
```

Format

A `numeric` vector containing 10,000 sorted values.

Examples

```
# "asymptotic" threshold value for 95% confidence
quantile(stepR::MRC.asymptotic.dyadic, .95)
```

neighbours *Neighbouring integers*

Description

Find integers within some radius of the given ones.

Usage

```
neighbours(k, x = 1:max(k), r = 0)
```

Arguments

k	integers within whose neighbourhood to look
x	allowed integers
r	radius within which to look

Value

Returns those integers in x which are at most r from some integer in k, i.e. the intersection of x with the union of the balls of radius r centred at the values of k. The return values are unique and sorted.

See Also

is.element, match, findInterval, stepcand

Examples

```
neighbours(c(10, 0, 5), r = 1)
neighbours(c(10, 0, 5), 0:15, r = 1)
```

parametricFamily *Parametric families*

Description

Overview about the supported parametric families (models). More details are given in section 5 of the vignette.

Details

The following parametric families (models and fitting methods) are available. Some of them have additional parameters that have to / can be specified in ....

`"gauss"` independent normal distributed variables with unknown mean but known, constant standard deviation given by the optional argument `sd`. Fits are obtained by the method SMUCE (*Frick et al.*, 2014) for independent normal distributed observations. Argument `sd` has to be a single, positive, finite `numeric`. If omitted it will be estimated by `sdrobnorm`. For `monteCarloSimulation` `sd == 1` will be used always. The observations argument `y` has to be a numeric vector with finite entries. The default interval system is `"all"` up to 1000 observations and `"dyaLen"` for more observations. Possible lengths are `1:length(y)`. The default penalty is `"sqrt"`. In `monteCarloSimulation` by default `n` random observations will be generated by `rnorm`.

`"hsmuce"` independent normal distributed variables with unknown mean and also unknown piecewise constant standard deviation as a nuisance parameter. Fits are obtained by the method HSMUCE (*Pein et al.*, 2017). No additional argument has to be given. The observations argument `y` has to be a numeric vector with finite entries. The default interval system is `"dyaPar"` and possible lengths are `2:length(y)`. The default penalty is `"weights"` which will automatically be converted to `"none"` in `computeStat` and `monteCarloSimulation`. In `monteCarloSimulation` by default `n` random observations will be generated by `rnorm`.

`"mDependentPS"` normal distributed variables with unknown mean and m-dependent errors with known covariance structure given either by the argument `covariances`, `correlations` or `filter`. Fits are obtained by the method SMUCE (*Frick et al.*, 2014) for m-dependent normal distributed observations using partial sum tests and minimizing the least squares distance (*Pein et al.*, 2017, (7) and (8)). If `correlations` or `filter` is used to specify the covariances an additional optional argument `sd` can be used to specify the constant standard deviation. If `covariances` is specified the arguments `correlations`, `filter` and `sd` will be ignored and if `correlations` is specified the argument `filter` will be ignored. The argument `covariances` has to be a finite numeric vector, m will be defined by `m = length(covariances) - 1`, giving the vector of covariances, i.e. the first element must be positive, the absolute value of every other element must be smaller than or equal to the first one and the last element should not be zero. The argument `correlation` has to be a finite numeric vector, m will be defined by `m = length(correlations) - 1`, giving the vector of correlations, i.e. the first element must be 1, the absolute value of every other element must be smaller than or equal to the first one and the last element should not be zero. Covariances will be calculated by `correlations * sd^2`. The argument `filter` has to be an object of class `lowpassFilter` from which the correlation vector will be obtained. The argument `sd` has to be a single, positive, finite `numeric`. If omitted it will be estimated by `sdrobnorm` with `lag = m + 1`. For `monteCarloSimulation` `sd == 1` will be used always. The observations argument `y` has to be a numeric vector with finite entries. The default interval system is `"dyaLen"` and possible lengths are `1:length(y)`. The default penalty is `"sqrt"`. In `monteCarloSimulation` by default `n` random observations will be generated by calculating the coefficients of the corresponding moving average process and generating random observations from it.

The family is selected via the `family` argument, providing the corresponding string, while additional parameters have to / can be specified in ... if any.

References

Frick, K., Munk, A., Sieling, H. (2014) Multiscale change-point inference. With discussion and rejoinder by the authors. *Journal of the Royal Statistical Society, Series B* **76**(3), 495–580.

Pein, F., Sieling, H., Munk, A. (2017) Heterogeneous change point inference. *Journal of the Royal Statistical Society, Series B*, **79**(4), 1207–1227.

Pein, F., Tecuapetla-Gómez, I., Schütte, O., Steinem, C., Munk, A. (2017) Fully-automatic multiresolution idealization for filtered ion channel recordings: flickering event detection. *arXiv*:1706.03671.

See Also

Distributions, sdrobnorm, rnorm

Examples

```
# parametric family "gauss": independent gaussian errors with constant variance
set.seed(1)
x <- seq(1 / 100, 1, 1 / 100)
y <- c(rnorm(50), rnorm(50, 2))
plot(x, y, pch = 16, col = "grey30", ylim = c(-3, 5))

# computation of SMUCE and its confidence statements
fit <- stepFit(y, x = x, alpha = 0.5, family = "gauss",
               jumpint = TRUE, confband = TRUE)
lines(fit, lwd = 3, col = "red", lty = "22")

# confidence intervals for the change-point locations
points(jumpint(fit), col = "red")
# confidence band
lines(confband(fit), lty = "22", col = "darkred", lwd = 2)

# "gauss" is default for family
identical(stepFit(y, x = x, alpha = 0.5, jumpint = TRUE, confband = TRUE), fit)
# missing sd is estimated by sdrobnorm
identical(stepFit(y, x = x, alpha = 0.5, family = "gauss", sd = sdrobnorm(y),
                  jumpint = TRUE, confband = TRUE), fit)

# parametric family "hsmuce": independent gaussian errors with also
# piecewise constant variance
# estimaton that is robust against variance changes
set.seed(1)
y <- c(rnorm(50, 0, 1), rnorm(50, 1, 0.2))
plot(x, y, pch = 16, col = "grey30", ylim = c(-2.5, 2))

# computation of HSMUCE and its confidence statements
fit <- stepFit(y, x = x, alpha = 0.5, family = "hsmuce",
               jumpint = TRUE, confband = TRUE)
lines(fit, lwd = 3, col = "red", lty = "22")

# confidence intervals for the change-point locations
points(jumpint(fit), col = "red")
```

```
# confidence band
lines(confband(fit), lty = "22", col = "darkred", lwd = 2)

# for comparison SMUCE
lines(stepFit(y, x = x, alpha = 0.5, jumpint = TRUE, confband = TRUE),
      lwd = 3, col = "blue", lty = "22")


# parametric family "mDependentPS": m dependent observations with known covariances
# observations are generated from a moving average process
set.seed(1)
y <- c(rep(0, 50), rep(2, 50)) +
  as.numeric(arima.sim(n = 100, list(ar = c(), ma = c(0.8, 0.5, 0.3)), sd = 0.5))
correlations <- as.numeric(ARMAacf(ar = c(), ma = c(0.8, 0.5, 0.3), lag.max = 3))
covariances <- 0.5^2 * correlations
plot(x, y, pch = 16, col = "grey30", ylim = c(-2, 4))

# computation of SMUCE for dependent observations with given covariances
fit <- stepFit(y, x = x, alpha = 0.5, family = "mDependentPS",
               covariances = covariances, jumpint = TRUE, confband = TRUE)
lines(fit, lwd = 3, col = "red", lty = "22")

# confidence intervals for the change-point locations
points(jumpint(fit), col = "red")
# confidence band
lines(confband(fit), lty = "22", col = "darkred", lwd = 2)

# for comparison SMUCE for independent gaussian errors
lines(stepFit(y, x = x, alpha = 0.5, jumpint = TRUE, confband = TRUE),
      lwd = 3, col = "blue", lty = "22")

# covariance structure can also be given by correlations and sd
identical(stepFit(y, x = x, alpha = 0.5, family = "mDependentPS",
                  correlations = correlations, sd = 0.5,
                  jumpint = TRUE, confband = TRUE), fit)

# if sd is missing it will be estimated by sdrobnorm
identical(stepFit(y, x = x, alpha = 0.5,family = "mDependentPS",
                  correlations = correlations, jumpint = TRUE, confband = TRUE),
          stepFit(y, x = x, alpha = 0.5, family = "mDependentPS",
                  correlations = correlations,
                  sd = sdrobnorm(y, lag = length(correlations)),
                  jumpint = TRUE, confband = TRUE))
```

`penalty` *Penalties*

Description

Overview about the supported penalties. More details are also given in section 3.2 of the vignette.

Details

The penalties (ways to balance different scales) can be divided into two groups: scale penalisation and balancing by weights. More precisely, the scale penalisations `"sqrt"`, `"log"` and `"none"` and balancing by weights called `"weights"` are available.

Let `T` be the unpenalised test statistic of the specified parametric family on an interval of length `l` and `nq` the number of observations used for the penalisation, typically the number of observations `n` but can also be chosen larger.

`"sqrt"` penalised statistic is `sqrt(2 * T) - sqrt(2 * log(exp(1) * nq / l))`. This penalisation is proposed in (Frick et al., 2014) and guarantees for most parametric families that the penalised multiscale statistic is asymptotically finite. This is not true for parametric family `"hsmuce"`. Hence, this penalisation is recommended and the default one for the parametric families `"gauss"` and `"mDependentPS"`, but not for `"hsmuce"`.

`"log"` penalised statistic is `T - log(exp(1) * nq / l)`. This penalisation is outdated and only still supported for comparisons.

`"none"` no penalisation, penalised statistic is equal to the unpenalised. Multiscale regression without a penalisation is not recommend.

`"weights"` critical values will be computed by weights, see section 3.2.2 in the vignette and (Pein et al., 2017, section 2) for more details. This penalty is recommend and the default one for the parametric family `"hsmuce"`, but can also be used for other families. Will be replaced by `"none"` in `computeStat` and `monteCarloSimulation`.

The penalisation is selected via the `penalty` argument providing the corresponding string. If `NULL` the default penalty of the specified parametric family will be used, see `parametricFamily` for which one this will be.

References

Frick, K., Munk, A., Sieling, H. (2014) Multiscale change-point inference. With discussion and rejoinder by the authors. *Journal of the Royal Statistical Society, Series B* **76**(3), 495–580.

Pein, F., Sieling, H., Munk, A. (2017) Heterogeneous change point inference. *Journal of the Royal Statistical Society, Series B*, **79**(4), 1207–1227.

See Also

parametricFamily, `critVal`

Examples

```
set.seed(1)
y <- c(rnorm(50), rnorm(50, 2))

# penalty "sqrt"
fit <- stepFit(y, alpha = 0.5, penalty = "sqrt", jumpint = TRUE, confband = TRUE)

# default for family "gauss"
identical(stepFit(y, alpha = 0.5, jumpint = TRUE, confband = TRUE), fit)
```

```
# penalty "weights"
!identical(stepFit(y, alpha = 0.5, penalty = "weights",
                   jumpint = TRUE, confband = TRUE), fit)

# penalty "weights" is default for parametric family "hsmuce"
# by default equal weights are chosen
identical(stepFit(y, alpha = 0.5, family = "hsmuce",
                  jumpint = TRUE, confband = TRUE),
          stepFit(y, alpha = 0.5, family = "hsmuce", penalty = "weights",
                  weights = rep(1 / 6, 6), jumpint = TRUE, confband = TRUE))

# different weights
!identical(stepFit(y, alpha = 0.5, family = "hsmuce", weights = 6:1 / sum(6:1),
                   jumpint = TRUE, confband = TRUE),
           stepFit(y, alpha = 0.5, family = "hsmuce", penalty = "weights",
                   weights = rep(1 / 6, 6), jumpint = TRUE, confband = TRUE))

# penalty "sqrt is default for parametric family "mDependentPS"
identical(stepFit(y, alpha = 0.5, family = "mDependentPS", covariances = c(1, 0.5),
                  jumpint = TRUE, confband = TRUE),
          stepFit(y, alpha = 0.5, family = "mDependentPS", covariances = c(1, 0.5),
                  penalty = "sqrt", jumpint = TRUE, confband = TRUE))
```

`sdrobnorm` *Robust standard deviation estimate*

Description

Robust estimation of the standard deviation of Gaussian data.

Usage

```
sdrobnorm(x, p = c(0.25, 0.75), lag = 1,
          supressWarningNA = FALSE, supressWarningResultNA = FALSE)
```

Arguments

`x`	a vector of numerical observations. `NA` entries will be removed with a warning. The warning can be supressed by setting `supressWarningNA` to `TRUE`. Other non finite values are not allowed
`p`	vector of two distinct probabilities
`lag`	a single integer giving the lag of the difference used, see `diff`, if a numeric is passed a small tolerance will be added and the value will be converted by `as.integer`
`supressWarningNA`	a single logical, if `TRUE` no warning will be given for `NA` entries in `x`

supressWarningResultNA
: a single logical, if TRUE no warning will be given if the result is NA

Details

Compares the difference between the estimated sample quantile corresponding to p after taking (lagged) differences) with the corresponding theoretical quantiles of Gaussian white noise to determine the standard deviation under a Gaussian assumption. If the data contain (few) jumps, this will (on average) be a slight overestimate of the true standard deviation.

This estimator has been inspired by (1.7) in (*Davies and Kovac*, 2001).

Value

Returns the estimate of the sample's standard deviation, i.e. a single non-negative numeric, NA if length(x) < lag + 2.

References

Davies, P. L., Kovac, A. (2001) Local extremes, runs, strings and multiresolution. *The Annals of Statistics* **29**, 1–65.

See Also

sd, diff, parametricFamily, family

Examples

```
# simulate data sample
y <- rnorm(100, c(rep(1, 50), rep(10, 50)), 2)
# estimate standard deviation
sdrobnorm(y)
```

---

smuceR *Piecewise constant regression with SMUCE*

Description

Computes the SMUCE estimator for one-dimensional data.

Deprecation warning: This function is deprecated, but still working, however, may be defunct in a future version. Please use instead the function stepFit. At the moment some families are supported by this function that are not supported by the current version of stepFit. They will be added in a future version. An example how to reproduce results is given below.

Usage

```
smuceR(y, x = 1:length(y), x0 = 2 * x[1] - x[2], q = thresh.smuceR(length(y)), alpha, r,
  lengths, family = c("gauss", "gaussvar", "poisson", "binomial"), param,
  jumpint = confband, confband = FALSE)
thresh.smuceR(v)
```

Arguments

y	a numeric vector containing the serial data
x	a numeric vector of the same length as y containing the corresponding sample points
x0	a single numeric giving the last unobserved sample point directly before sampling started
q	threshold value, by default chosen automatically according to Frick et al.~(2013)
alpha	significance level; if set to a value in (0,1), q is chosen as the corresponding quantile of the asymptotic (if r is not given) null distribution (and any value specified for q is silently ignored)
r	numer of simulations; if specified along alpha, q is chosen as the corresponding quantile of the simulated null distribution
lengths	length of intervals considered; by default up to a sample size of 1000 all lengths, otherwise only dyadic lengths
family, param	specifies distribution of data, see family
jumpint	logical (FALSE by default), indicates if confidence sets for change-points should be computed
confband	logical, indicates if a confidence band for the piecewise-continuous function should be computed
v	number of data points

Value

For smuceR, an object of class stepfit that contains the fit; if jumpint == TRUE function jumpint allows to extract the 1 - alpha confidence interval for the jumps, if confband == TRUE function confband allows to extract the 1 - alpha confidence band.

For thresh.smuceR, a precomputed threshhold value, see reference.

References

Frick, K., Munk, A., and Sieling, H. (2014) Multiscale change-point inference. With discussion and rejoinder by the authors. *Journal of the Royal Statistical Society, Series B* **76**(3), 495–580.

Futschik, A., Hotz, T., Munk, A. Sieling, H. (2014) Multiresolution DNA partitioning: statistical evidence for segments. *Bioinformatics*, **30**(16), 2255–2262.

See Also

stepFit, stepbound, bounds, family, MRC.asymptotic, sdrobnorm, stepfit

Examples

```
y <- rnorm(100, c(rep(0, 50), rep(1, 50)), 0.5)

# fitted function, confidence intervals, and confidence band by stepFit
all.equal(fitted(smuceR(y, q = 1)), fitted(stepFit(y, q = 1)))
```

```
all.equal(fitted(smuceR(y, alpha = 0.5)),
          fitted(stepFit(y, q = as.numeric(quantile(stepR::MRC.1000, 0.5)))))
all.equal(fitted(smuceR(y)), fitted(stepFit(y, q = thresh.smuceR(length(y)))))

all.equal(jumpint(smuceR(y, q = 1, jumpint = TRUE)),
          jumpint(stepFit(y, q = 1, jumpint = TRUE)))
all.equal(confband(smuceR(y, q = 1, confband = TRUE)),
          confband(stepFit(y, q = 1, confband = TRUE)),
          check.attributes = FALSE)


# simulate poisson data with two levels
y <- rpois(100, c(rep(1, 50), rep(4, 50)))
# compute fit, q is chosen automatically
fit <- smuceR(y, family="poisson", confband = TRUE)
# plot result
plot(y)
lines(fit)
# plot confidence intervals for jumps on axis
points(jumpint(fit), col="blue")
# confidence band
lines(confband(fit), lty=2, col="blue")

# simulate binomial data with two levels
y <- rbinom(200,3,rep(c(0.1,0.7),c(110,90)))
# compute fit, q is the 0.9-quantile of the (asymptotic) null distribution
fit <- smuceR(y, alpha=0.1, family="binomial", param=3, confband = TRUE)
# plot result
plot(y)
lines(fit)
# plot confidence intervals for jumps on axis
points(jumpint(fit), col="blue")
# confidence band
lines(confband(fit), lty=2, col="blue")
```

`stepblock` *Step function*

Description

Constructs an object containing a step function sampled over finitely many values.

Usage

```
stepblock(value, leftEnd = c(1, rightEnd[-length(rightEnd)] + 1), rightEnd, x0 = 0)
## S3 method for class 'stepblock'
x[i, j, drop = if(missing(i)) TRUE else if(missing(j)) FALSE else length(j) == 1, ...]
## S3 method for class 'stepblock'
print(x, ...)
## S3 method for class 'stepblock'
```

```
plot(x, type = "c", xlab = "x", ylab = "y", main = "Step function", sub = NULL, ...)
## S3 method for class 'stepblock'
lines(x, type = "c", ...)
```

Arguments

value	a numeric vector containing the fitted values for each block; its length gives the number of blocks
leftEnd	a numeric vector of the same length as value containing the left end of each block
rightEnd	a numeric vector of the same length as value containing the right end of each block
x0	a single numeric giving the last unobserved sample point directly before sampling started, i.e. before leftEnd[1]
x	the object
i, j, drop	see [.data.frame
type	"c" to plot jumps in the middle between the end of the previous block (or x0) and the beginning of the following block; "e" to jump at the end of the previous block; "b" to jump at the beginning of the following block; capital letters also plot points
xlab, ylab, main, sub	see plot.default
...	for generic methods only

Value

For stepblock an object of class stepblock, i.e. a data.frame with columns value, leftEnd and rightEnd and attribute x0.

Note

For the purposes of this package step functions are taken to be left-continuous, i.e. the function jumps **after** the rightEnd of a block.

However, step functions are usually sampled at a discrete set of points so that the exact position of the jump is unknown, except that it has to occur before the next sampling point; this is expressed in the implementation by the specification of a leftEnd **within** the block so that every rightEnd and leftEnd is a sampling point (or the boundary of the observation window), there is no sampling point between one block's rightEnd and the following block's leftEnd, while the step function is constant at least on the closed interval with boundary leftEnd, rightEnd.

See Also

step, stepfit, family, [.data.frame, plot, lines

Examples

```
# step function consisting of 3 blocks: 1 on (0, 3]; 2 on (3, 6], 0 on (6, 8]
# sampled on the integers 1:10
f <- stepblock(value = c(1, 2, 0), rightEnd = c(3, 6, 8))
f
# show different plot types
plot(f, type = "C")
lines(f, type = "E", lty = 2, col = "red")
lines(f, type = "B", lty = 3, col = "blue")
legend("bottomleft", legend = c("C", "E", "B"), lty = 1:3, col = c("black", "red", "blue"))
```

---

stepbound *Jump estimation under restrictions*

Description

Computes piecewise constant maximum likelihood estimators with minimal number of jumps under given restrictions on subintervals.

Deprecation warning: This function is a help function for smuceR and jsmurf and may be removed when these function will be removed.

Usage

```
stepbound(y, bounds, ...)
## Default S3 method:
stepbound(y, bounds, x = 1:length(y), x0 = 2 * x[1] - x[2],
 max.cand = NULL, family = c("gauss", "gaussvar", "poisson", "binomial", "gaussKern"),
  param = NULL, weights = rep(1, length(y)), refit = y,
  jumpint = confband, confband = FALSE, ...)
## S3 method for class 'stepcand'
stepbound(y, bounds, refit = TRUE, ...)
```

Arguments

y	a vector of numerical observations
bounds	bounds on the value allowed on intervals; typically computed with bounds
x	a numeric vector of the same length as y containing the corresponding sample points
x0	a single numeric giving the last unobserved sample point directly before sampling started
max.cand, weights	see stepcand
family, param	specifies distribution of data, see family
refit	logical, for family = "gaussKern"; determines whether a fit taken the filter kernel into account will be computed at the end

jumpint	logical (FALSE by default), indicates if confidence sets for jumps should be computed
confband	logical, indicates if a confidence band for the piecewise-continuous function should be computed
...	arguments to be passed to generic methods

Value

An object of class stepfit that contains the fit; if jumpint == TRUE function jumpint allows to extract the confidence interval for the jumps, if confband == TRUE function confband allows to extract the confidence band.

References

Frick, K., Munk, A., and Sieling, H. (2014) Multiscale change-point inference. With discussion and rejoinder by the authors. *Journal of the Royal Statistical Society, Series B* **76**(3), 495–580.

Hotz, T., Schütte, O., Sieling, H., Polupanow, T., Diederichsen, U., Steinem, C., and Munk, A. (2013) Idealizing ion channel recordings by a jump segmentation multiresolution filter. *IEEE Transactions on NanoBioscience* **12**(4), 376–386.

See Also

bounds, smuceR, jsmurf, stepsel, stepfit, jumpint, confband

Examples

```
# simulate poisson data with two levels
y <- rpois(100, c(rep(1, 50), rep(4, 50)))
# compute bounds
b <- bounds(y, penalty="len", family="poisson", q=4)
# fit step function to bounds
sb <- stepbound(y, b, family="poisson", confband=TRUE)
plot(y)
lines(sb)
# plot confidence intervals for jumps on axis
points(jumpint(sb), col="blue")
# confidence band
lines(confband(sb), lty=2, col="blue")
```

---

stepcand *Forward selection of candidate jumps*

Description

Find candidates for jumps in serial data by forward selection.

Usage

```
stepcand(y, x = 1:length(y), x0 = 2 * x[1] - x[2], max.cand = NULL,
 family = c("gauss", "gaussvar", "poisson", "binomial", "gaussKern"), param = NULL,
  weights = rep(1, length(y)), cand.radius = 0)
```

Arguments

y	a numeric vector containing the serial data
x	a numeric vector of the same length as y containing the corresponding sample points
x0	a single numeric giving the last unobserved sample point directly before sampling started
max.cand	single integer giving the maximal number of blocks to find; defaults to using all data (note: there will be one block more than the number of jumps
family	distribution of the errors, either "gauss", "poisson" or "binomial"; "gaussInhibit" is like "gauss" forbids jumps getting close together or to the ends in steppath.stepcand, "gaussInhibitBoth" already forbids this in stepcand (not recommended)
param	additional parameters specifying the distribution of the errors; the number of trials for family "binomial"; for gaussInhibit and gaussInhibitBoth a numeric of length 3 with components "start", "middle" and "end" preventing the first jump from getting closer to x0 than the "start" value, any two jumps from getting closer than the "middle" value, and the last jump from geting closer than the "end" value to the end, all distances measured by weights (cf. example below)
weights	a numeric vector of the same length as y containing non-negative weights
cand.radius	a non-negative integer: adds for each candidate found all indices that are at most cand.radius away

Value

An object of class stepcand extending class stepfit such that it can be used as an input to steppath.stepcand: additionally contains columns

cumSum	The cumulative sum of x up to rightEnd.
cumSumSq	The cumulative sum of squares of x up to rightEnd (for family = "gauss").
cumSumWe	The cumulative sum of weights up to rightEnd.
improve	The improvement this jump brought about when it was selected.

See Also

steppath, stepfit, family

Examples

```
# simulate 5 blocks (4 jumps) within a total of 100 data points
b <- c(sort(sample(1:99, 4)), 100)
f <- rep(rnorm(5, 0, 4), c(b[1], diff(b)))
rbind(b = b, f = unique(f), lambda = exp(unique(f) / 10) * 20)
# add gaussian noise
x <- f + rnorm(100)
# find 10 candidate jumps
stepcand(x, max.cand = 10)
# for poisson observations
y <- rpois(100, exp(f / 10) * 20)
# find 10 candidate jumps
stepcand(y, max.cand = 10, family = "poisson")
# for binomial observations
size <- 10
z <- rbinom(100, size, pnorm(f / 10))
# find 10 candidate jumps
stepcand(z, max.cand = 10, family = "binomial", param = size)
```

---

stepFit *Piecewise constant multiscale inference*

Description

Computes the multiscale regression estimator, see (3.1) in the vignette, and allows for confidence statements, see section 3 in the vignette. It implements the estimators SMUCE and HSMUCE as well as their confidence intervals and bands.
If `q == NULL` a Monte-Carlo simulation is required for computing critical values. Since a Monte-Carlo simulation lasts potentially much longer (up to several hours or days if the number of observations is in the millions) than the main calculations, this package saves them by default in the workspace and on the file system such that a second call requiring the same Monte-Carlo simulation will be much faster. For more details, in particular to which arguments the Monte-Carlo simulations are specific, see Section *Storing of Monte-Carlo simulations* below. Progress of a Monte-Carlo simulation can be reported by the argument `messages` and the saving can be controlled by the argument `option`, both can be specified in `...` and are explained in `monteCarloSimulation` and `critVal`, respectively.

Usage

```
stepFit(y, q = NULL, alpha = NULL, x = 1:length(y), x0 = 2 * x[1] - x[2],
        family = NULL, intervalSystem = NULL, lengths = NULL, confband = FALSE,
        jumpint = confband, ...)
```

Arguments

y	a numeric vector containing the observations

`q` either `NULL`, then the vector of critical values at level `alpha` will be computed from a Monte-Carlo simulation, or a numeric giving the global quantile or a numeric vector giving the vector of critical values. Either `q` or `alpha` must be given. Otherwise, `alpha == 0.5` is chosen with a `warning`. This argument will be passed to `critVal` to obtain the needed critical values. Additional parameters for the computation of `q` can be specified in `...`, for more details see the documentation of `critVal`. Please note that by default the Monte-Carlo simulation will be saved in the workspace and on the file system, for more details see Section *Storing of Monte-Carlo simulations* below

`alpha` a probability, i.e. a single numeric between 0 and 1, giving the significance level. Its choice is a trade-off between data fit and parsimony of the estimator. In other words, this argument balances the risks of missing change-points and detecting additional artefacts. For more details on this choice see (Frick et al., 2014, section 4) and (Pein et al., 2017, section 3.4). Either `q` or `alpha` must be given. Otherwise, `alpha == 0.5` is chosen with a `warning`

`x` a numeric vector of the same length as `y` containing the corresponding sample points

`x0` a single numeric giving the last unobserved sample point directly before sampling started

`family` a string specifying the assumed parametric family, for more details see parametricFamily, currently `"gauss"`, `"hsmuce"` and `"mDependentPS"` are supported. By default (`NULL`) `"gauss"` is assumed

`intervalSystem` a string giving the used interval system, either `"all"` for all intervals, `"dyaLen"` for all intervals of dyadic length or `"dyaPar"` for the dyadic partition, for more details see intervalSystem. By default (`NULL`) the default interval system of the specified parametric family will be used, which one this will be is described in `parametricFamily`

`lengths` an integer vector giving the set of lengths, i.e. only intervals of these lengths will be considered. Note that not all lengths are possible for all interval systems and for all parametric families, see intervalSystem and parametricFamily, respectively, to see which ones are allowed. By default (`NULL`) all lengths that are possible for the specified `intervalSystem` and for the specified parametric family will be used

`confband` single `logical`, indicates if a confidence band for the piecewise-continuous function should be computed

`jumpint` single `logical`, indicates if confidence sets for change-points should be computed

`...` there are two groups of further arguments:

1. further parameters of the parametric family. Depending on argument `family` some might be required, but others might be optional, please see parametricFamily for more details,
2. further parameters that will be passed to `critVal`. `critVal` will be called automatically with the number of observations `n = length(y)`, the arguments `family`, `intervalSystem`, `lengths`, `q` and `output` set. For these arguments no user interaction is required and possible, all other arguments of `critVal` can be passed additionally

Value

An object of class stepfit that contains the fit. If jumpint == TRUE function jumpint allows to extract the 1 - alpha confidence interval for the jumps. If confband == TRUE function confband allows to extract the 1 - alpha confidence band.

Storing of Monte-Carlo simulations

If q == NULL a Monte-Carlo simulation is required for computing critical values. Since a Monte-Carlo simulation lasts potentially much longer (up to several hours or days if the number of observations is in the millions) than the main calculations, this package offers multiple possibilities for saving and loading the simulations. Progress of a simulation can be reported by the argument messages which can be specified in ... and is explained in the documentation of monteCarloSimulation. Each Monte-Carlo simulation is specific to the number of observations, the parametric family (including certain parameters, see parametricFamily) and the interval system, and for simulations of class "MCSimulationMaximum", additionally, to the set of lengths and the used penalty. Monte-Carlo simulations can also be performed for a (slightly) larger number of observations n_q given in the argument nq in ... and explained in the documentation of critVal, which avoids extensive resimulations for only a little bit varying number of observations. Simulations can either be saved in the workspace in the variable critValStepRTab or persistently on the file system for which the package R.cache is used. Moreover, storing in and loading from variables and RDS files is supported. Finally, a pre-simulated collection of simulations can be accessed by installing the package stepRdata available from http://www.stochastik.math.uni-goettingen.de/stepRdata_1.0-0.tar.gz. The simulation, saving and loading can be controlled by the argument option which can be specified in ... and is explained in the documentation of critVal. By default simulations will be saved in the workspace and on the file system. For more details and for how simulation can be removed see Section *Simulating, saving and loading of Monte-Carlo simulations* in critVal.

References

Frick, K., Munk, A., Sieling, H. (2014) Multiscale change-point inference. With discussion and rejoinder by the authors. *Journal of the Royal Statistical Society, Series B* **76**(3), 495–580.

Pein, F., Sieling, H., Munk, A. (2017) Heterogeneous change point inference. *Journal of the Royal Statistical Society, Series B*, **79**(4), 1207–1227.

See Also

critVal, penalty, parametricFamily, intervalSystem, monteCarloSimulation

Examples

```
# generate random observations
y <- c(rnorm(50), rnorm(50, 1))
x <- seq(0.01, 1, 0.01)
plot(x, y, pch = 16, col = "grey30", ylim = c(-3, 4))

# computation of SMUCE and its confidence statements
fit <- stepFit(y, x = x, alpha = 0.5, jumpint = TRUE, confband = TRUE)
lines(fit, lwd = 3, col = "red", lty = "22")
```

```
# confidence intervals for the change-point locations
points(jumpint(fit), col = "red")
# confidence band
lines(confband(fit), lty = "22", col = "darkred", lwd = 2)

# higher significance level for larger detection power, but less confidence
stepFit(y, x = x, alpha = 0.99, jumpint = TRUE, confband = TRUE)

# smaller significance level for the small risk that the number of
# change-points is overestimated with probability not more than 5%,
# but smaller detection power
stepFit(y, x = x, alpha = 0.05, jumpint = TRUE, confband = TRUE)

# different interval system, lengths, penalty and given parameter sd
stepFit(y, x = x, alpha = 0.5, intervalSystem = "dyaLen",
        lengths = c(1L, 2L, 4L, 8L), penalty = "weights",
        weights = c(0.4, 0.3, 0.2, 0.1), sd = 0.5,
        jumpint = TRUE, confband = TRUE)

# with given q
identical(stepFit(y, x = x, q = critVal(100L, alpha = 0.5),
                  jumpint = TRUE, confband = TRUE), fit)
identical(stepFit(y, x = x, q = critVal(100L, alpha = 0.5, output = "value"),
                  jumpint = TRUE, confband = TRUE), fit)

# the above calls saved and (attempted to) load Monte-Carlo simulations and
# simulated them for nq = 128 observations
# in the following call no saving, no loading and simulation for n = 100
# observations is required, progress of the simulation will be reported
stepFit(y, x = x, alpha = 0.5, jumpint = TRUE, confband = TRUE,
        messages = 1000L, options = list(simulation = "vector",
                                         load = list(), save = list()))

# with given stat to compute q
stat <- monteCarloSimulation(n = 128L)
identical(stepFit(y, x = x, alpha = 0.5, stat = stat,
                  jumpint = TRUE, confband = TRUE),
          stepFit(y, x = x, alpha = 0.5, jumpint = TRUE, confband = TRUE,
                  options = list(load = list())))
```

`stepfit` *Fitted step function*

Description

Constructs an object containing a step function fitted to some data.

Usage

```
stepfit(cost, family, value, param = NULL, leftEnd, rightEnd, x0,
  leftIndex = leftEnd, rightIndex = rightEnd)
## S3 method for class 'stepfit'
x[i, j, drop = if(missing(i)) TRUE else
  if(missing(j)) FALSE else length(j) == 1, refit = FALSE]
## S3 method for class 'stepfit'
print(x, ...)
## S3 method for class 'stepfit'
plot(x, dataspace = TRUE, ...)
## S3 method for class 'stepfit'
lines(x, dataspace = TRUE, ...)
## S3 method for class 'stepfit'
fitted(object, ...)
## S3 method for class 'stepfit'
residuals(object, y, ...)
## S3 method for class 'stepfit'
logLik(object, df = NULL, nobs = object$rightIndex[nrow(object)], ...)
```

Arguments

cost	the value of the cost-functional used for the fit: RSS for family gauss, log-likelihood (up to a constant) for families poisson and binomial
family	distribution of the errors, either "gauss", "poisson" or "binomial"
value	a numeric vector containing the fitted values for each block; its length gives the number of blocks
param	additional paramters specifying the distribution of the errors, the number of trials for family "binomial"
leftEnd	a numeric vector of the same length as value containing the left end of each block
rightEnd	a numeric vector of the same length as value containing the left end of each block
x0	a single numeric giving the last unobserved sample point directly before sampling started, i.e. before leftEnd[0]
leftIndex	a numeric vector of the same length as value containing the index of the sample points corresponding to the block's left end, cf. stepcand
rightIndex	a numeric vector of the same length as value containing the index of the sample points corresponding to the block's right end, cf. stepcand
x, object	the object
y	a numeric vector containing the data with which to compare the fit
df	the number of estimated parameters: by default the number of blocks for families poisson and binomial, one more (for the variance) for family gauss
nobs	the number of observations used for estimating
...	for generic methods only

i, j, drop — see "[.data.frame"

refit — logical; determines whether the function will be refitted after subselection, i.e. whether the selection should be interpreted as a fit with fewer jumps); in that case, for family = "gaussKern", refit needs to be set to the original data, i.e. y

dataspace — logical determining whether the expected value should be plotted instead of the fitted parameter value, useful e.g. for family = "binomial", where it will plot the fitted success probability times the number of trials per observation

Value

stepfit — an object of class stepfit which extends stepblock, additionally containing attributes cost, family and param, as well as columns leftIndex and rightIndex

[.stepfit — an object of class stepfit which contains the selected subset

fitted.stepfit — a numeric vector of length rightIndex[length(rightIndex)] giving the fit at the original sample points

residuals.stepfit — a numeric vector of length rightIndex[length(rightIndex)] giving the residuals at the original sample points

logLik.stepfit — an object of class logLik giving the likelihood of the data given this fit, e.g. for use with AIC and stepsel; this will (incorrectly) treat family = "gaussKern" as if it were fitted with family = "gauss"

plot.stepfit, plot.stepfit — the corresponding functions for stepblock are called

See Also

stepblock, stepbound, steppath, stepsel, family, "[.data.frame", fitted, residuals, logLik, AIC

Examples

```
# simulate 5 blocks (4 jumps) within a total of 100 data points
b <- c(sort(sample(1:99, 4)), 100)
p <- rep(runif(5), c(b[1], diff(b))) # success probabilities
# binomial observations, each with 10 trials
y <- rbinom(100, 10, p)
# find solution with 5 blocks
fit <- steppath(y, family = "binomial", param = 10)[[5]]
plot(y, ylim = c(0, 10))
lines(fit, col = "red")
# residual diagnostics for Gaussian data
yg <- rnorm(100, qnorm(p), 1)
fitg <- steppath(yg)[[5]]
plot(yg, ylim = c(0, 10))
lines(fitg, col = "red")
plot(resid(fitg, yg))
qqnorm(resid(fitg, yg))
```

steppath *Solution path of step-functions*

Description

Find optimal fits with step-functions having jumps at given candidate positions for all possible subset sizes.

Usage

```
steppath(y, ..., max.blocks)
## Default S3 method:
steppath(y, x = 1:length(y), x0 = 2 * x[1] - x[2], max.cand = NULL,
 family = c("gauss", "gaussvar", "poisson", "binomial", "gaussKern"), param = NULL,
  weights = rep(1, length(y)), cand.radius = 0, ..., max.blocks = max.cand)
## S3 method for class 'stepcand'
steppath(y, ..., max.blocks = sum(!is.na(y$number)))
## S3 method for class 'steppath'
x[[i]]
## S3 method for class 'steppath'
length(x)
## S3 method for class 'steppath'
print(x, ...)
## S3 method for class 'steppath'
logLik(object, df = NULL, nobs = object$cand$rightIndex[nrow(object$cand)], ...)
```

Arguments

for steppath:

y	either an object of class stepcand for steppath.stepcand or a numeric vector containing the serial data for steppath.default
x, x0, max.cand, family, param, weights, cand.radius	for steppath.default which calls stepcand; see there
max.blocks	single integer giving the maximal number of blocks to find; defaults to number of candidates (note: there will be one block more than the number of jumps
...	for generic methods only

for methods on a steppath object x or object:

object	the object
i	if this is an integer returns the fit with i blocks as an object of class stepcand, else the standard behaviour of a list
df	the number of estimated parameters: by default the number of blocks for families poisson and binomial, one more (for the variance) for family gauss
nobs	the number of observations used for estimating

Value

For steppath an object of class steppath, i.e. a list with components

path	A list of length length(object) where the ith element contains the best fit by a step-function having i-1 jumps (i.e. i blocks), given by the candidates indices
cost	A numeric vector of length length(object) giving the value of the cost functional corresponding to the solutions.
cand	An object of class stepcand giving the candidates among which the jumps were selected.

[[.steppath returns the fit with i blocks as an object of class stepfit; length.steppath the maximum number of blocks for which a fit has been computed. logLik.stepfit returns an object of class logLik giving the likelihood of the data given the fits corresponding to cost, e.g. for use with AIC.

References

Friedrich, F., Kempe, A., Liebscher, V., Winkler, G. (2008) Complexity penalized M-estimation: fast computation. *Journal of Computational and Graphical Statistics* **17**(1), 201–224.

See Also

stepcand, stepfit, family, logLik, AIC

Examples

```
# simulate 5 blocks (4 jumps) within a total of 100 data points
b <- c(sort(sample(1:99, 4)), 100)
f <- rep(rnorm(5, 0, 4), c(b[1], diff(b)))
# add Gaussian noise
x <- f + rnorm(100)
# find 10 candidate jumps
cand <- stepcand(x, max.cand = 10)
cand
# compute solution path
path <- steppath(cand)
path
plot(x)
lines(path[[5]], col = "red")
# compare result having 5 blocks with truth
fit <- path[[5]]
fit
logLik(fit)
AIC(logLik(fit))
cbind(fit, trueRightEnd = b, trueLevel = unique(f))
# for poisson observations
y <- rpois(100, exp(f / 10) * 20)
# compute solution path, compare result having 5 blocks with truth
cbind(steppath(y, max.cand = 10, family = "poisson")[[5]],
  trueRightEnd = b, trueIntensity = exp(unique(f) / 10) * 20)
# for binomial observations
```

```
size <- 10
z <- rbinom(100, size, pnorm(f / 10))
# compute solution path, compare result having 5 blocks with truth
cbind(steppath(z, max.cand = 10, family = "binomial", param = size)[[5]],
  trueRightEnd = b, trueIntensity = pnorm(unique(f) / 10))
# an example where stepcand is not optimal but indices found are close to optimal ones
blocks <- c(rep(0, 9), 1, 3, rep(1, 9))
blocks
stepcand(blocks, max.cand = 3)[,c("rightEnd", "value", "number")]
# erroneously puts the "1" into the right block in the first step
steppath(blocks)[[3]][,c("rightEnd", "value")]
# putting the "1" in the middle block is optimal
steppath(blocks, max.cand = 3, cand.radius = 1)[[3]][,c("rightEnd", "value")]
# also looking in the 1-neighbourhood remedies the problem
```

`stepsel` *Automatic selection of number of jumps*

Description

Select the number of jumps.

Usage

```
stepsel(path, y, type = c("MRC", "AIC", "BIC"), ...)
stepsel.MRC(path, y, q, alpha = 0.05, r = ceiling(50 / min(alpha, 1 - alpha)),
  lengths = if(attr(path$cand, "family") == "gaussKern")
   2^(floor(log2(length(y))):ceiling(log2(length(attr(path$cand, "param")$kern)))) else
    2^(floor(log2(length(y))):0),
 penalty = c("none", "log", "sqrt"), name = if(attr(path$cand, "family") == "gaussKern")
    ".MRC.ktable" else ".MRC.table",
  pos = .MCstepR)
stepsel.AIC(path, ...)
stepsel.BIC(path, ...)
```

Arguments

`path`	an object of class `steppath`
`y`	for `type=MRC` only: a numeric vector containing the serial data
`type`	how to select, dispatches specific method
`...`	further argument passed to specific method
`q, alpha, r, lengths, penalty, name, pos`	see `bounds`

Value

A single integer giving the number of blocks selected, with `attr`ibute `crit` containing the values of the criterion (MRC / AIC / BIC) for each fit in the path.

Note

To obtain the threshold described in Boysen et al.~(2009, Theorem~5), set `q=(1+delta) * sdrobnorm(y) * sqrt(2*length(y))` for some positive `delta` and `penalty="none"`.

References

Boysen, L., Kempe, A., Liebscher, V., Munk, A., Wittich, O. (2009) Consistencies and rates of convergence of jump-penalized least squares estimators. *The Annals of Statistics* **37**(1), 157–183.

Yao, Y.-C. (1988) Estimating the number of change-points via Schwarz' criterion. *Statistics & Probability Letters* **6**, 181–189.

See Also

`steppath`, `stepfit`, `family`, `stepbound`

Examples

```
# simulate 5 blocks (4 jumps) within a total of 100 data points
b <- c(sort(sample(1:99, 4)), 100)
f <- rep(rnorm(5, 0, 4), c(b[1], diff(b)))
rbind(b = b, f = unique(f))
# add gaussian noise
y <- f + rnorm(100)
# find 10 candidate jumps
path <- steppath(y, max.cand = 10)
# select number of jumps by simulated MRC with sqrt-penalty
# thresholded with positive delta, and by BIC
sel.MRC <- stepsel(path, y, "MRC", alpha = 0.05, r = 1e2, penalty = "sqrt")
sel.MRC
delta <- .1
sel.delta <- stepsel(path, y, "MRC",
  q = (1 + delta) * sdrobnorm(y) * sqrt(2 * length(y)), penalty = "none")
sel.delta
sel.BIC <- stepsel(path, type="BIC")
sel.BIC
# compare results with truth
fit.MRC <- path[[sel.MRC]]
as.data.frame(fit.MRC)
as.data.frame(path[[sel.delta]])
as.data.frame(path[[sel.BIC]])
```

---

testSmallScales *Test Small Scales*

Description

For developers only; users should look at the function `improveSmallScales` in the CRAN package `clampSeg`. Implements the second step of HILDE (*Pein et al.*, 2020, Section III-B) in which an initial fit is tested for missed short events.

Usage

```
.testSmallScales(data, family, lengths = NULL, q, alpha, ...)
```

Arguments

data	a numeric vector containing the observations
family	a string specifying the assumed parametric family, currently "LR" and "2Param" are supported
lengths	an integer vector giving the set of lengths, i.e. only intervals of these lengths will be considered. By default (NULL) 1:20 will be used for parametric family "LR" and 1:65 will be used for parametric family "2Param"
q	either NULL, then the vector of critical values at level alpha will be computed from a Monte-Carlo simulation or a numeric vector giving the vector of critical values. Either q or alpha must be given. Otherwise, alpha == 0.5 is chosen with a warning. This argument will be passed to critVal to obtain the needed critical values. Additional parameters for the computation of q can be specified in ..., for more details see the documentation of critVal. Please note that by default the Monte-Carlo simulation will be saved in the workspace and on the file system, for more details see Section *Storing of Monte-Carlo simulations* below
alpha	a probability, i.e. a single numeric between 0 and 1, giving the significance level. Its choice is a trade-off between data fit and parsimony of the estimator. In other words, this argument balances the risks of missing change-points and detecting additional artefacts. For more details on this choice see (Frick et al., 2014, section 4) and (Pein et al., 2017, section 3.4). Either q or alpha must be given. Otherwise, alpha == 0.5 is chosen with a warning
...	there are two groups of further arguments: 1. further parameters of the parametric family, 2. further parameters that will be passed to critVal. critVal will be called automatically with the number of observations n = length(y), the arguments family, intervalSystem, lengths, q and output set. For these arguments no user interaction is required and possible, all other arguments of critVal can be passed additionally

Value

a list with entries jumps, addLeft, addRight, noDeconvolution, data, q

References

Pein, F., Bartsch, A., Steinem, C., and Munk, A. (2020) Heterogeneous idealization of ion channel recordings - Open channel noise. Submitted.

`transit` *TRANSIT algorithm for detecting jumps*

Description

Reimplementation of VanDongen's algorithm for detecting jumps in ion channel recordings.

Deprecation warning: This function is mainly used for patchlamp recordings and may be transferred to a specialised package.

Usage

```
transit(y, x = 1:length(y), x0 = 2 * x[1] - x[2], sigma.amp = NA, sigma.slope = NA,
  amp.thresh = 3, slope.thresh = 2, rel.amp.n = 3, rel.amp.thresh = 4,
  family = c("gauss", "gaussKern"), param = NULL, refit = FALSE)
```

Arguments

`y`	a numeric vector containing the serial data
`sigma.amp`	amplitude (i.e. raw data within block) standard deviation; estimated using `sdrobnorm` if omitted
`sigma.slope`	slope (i.e. central difference within block) standard deviation; estimated using `sdrobnorm` if omitted
`amp.thresh`	amplitude threshold
`slope.thresh`	slope threshold
`rel.amp.n`	relative amplitude threshold will be used for blocks with no more datapoints than this
`rel.amp.thresh`	relative amplitude threshold
`x`	a numeric vector of the same length as `y` containing the corresponding sample points
`x0`	a single numeric giving the last unobserved sample point directly before sampling started
`family, param`	specifies distribution of data, see family
`refit`	should the values for `family = "gaussKern"` be obtained by fitting in the end (otherwise they are meaningless)

Value

Returns an object of class `stepfit` which encodes the jumps and corresponding mean values.

Note

Only central, no forward differences have been used in this implementation. Moreover, the standard deviations will be estimated by `sdrobnorm` if omitted (respecting the filter's effect if applicable).

References

VanDongen, A. M. J. (1996) A new algorithm for idealizing single ion channel data containing multiple unknown conductance levels. *Biophysical Journal* **70**(3), 1303–1315.

See Also

stepfit, sdrobnorm, jsmurf, stepbound, steppath

Examples

```
# estimating step-functions with Gaussian white noise added
# simulate a Gaussian hidden Markov model of length 1000 with 2 states
# with identical transition rates 0.01, and signal-to-noise ratio 2
sim <- contMC(1e3, 0:1, matrix(c(0, 0.01, 0.01, 0), 2), param=1/2)
plot(sim$data, cex = 0.1)
lines(sim$cont, col="red")
# maximum-likelihood estimation under multiresolution constraints
fit.MRC <- smuceR(sim$data$y, sim$data$x)
lines(fit.MRC, col="blue")
# choose number of jumps using BIC
path <- steppath(sim$data$y, sim$data$x, max.blocks=1e2)
fit.BIC <- path[[stepsel.BIC(path)]]
lines(fit.BIC, col="green3", lty = 2)

# estimate after filtering
# simulate filtered ion channel recording with two states
set.seed(9)
# sampling rate 10 kHz
sampling <- 1e4
# tenfold oversampling
over <- 10
# 1 kHz 4-pole Bessel-filter, adjusted for oversampling
cutoff <- 1e3
df.over <- dfilter("bessel", list(pole=4, cutoff=cutoff / sampling / over))
# two states, leaving state 1 at 10 Hz, state 2 at 20 Hz
rates <- rbind(c(0, 10), c(20, 0))
# simulate 0.5 s, level 0 corresponds to state 1, level 1 to state 2
# noise level is 0.3 after filtering
Sim <- contMC(0.5 * sampling, 0:1, rates, sampling=sampling, family="gaussKern",
  param = list(df=df.over, over=over, sd=0.3))
plot(Sim$data, pch = ".")
lines(Sim$discr, col = "red")
# fit under multiresolution constraints using filter corresponding to sample rate
df <- dfilter("bessel", list(pole=4, cutoff=cutoff / sampling))
Fit.MRC <- jsmurf(Sim$data$y, Sim$data$x, param=df, r=1e2)
lines(Fit.MRC, col = "blue")
# fit using TRANSIT
Fit.trans <- transit(Sim$data$y, Sim$data$x)
lines(Fit.trans, col = "green3", lty=2)
```

Index

∗ **datasets**
MRC.1000, 39
MRC.asymptotic, 40
MRC.asymptotic.dyadic, 40
∗ **distribution**
family, 28
parametricFamily, 41
∗ **math**
BesselPolynomial, 8
∗ **nonparametric**
bounds, 9
compareBlocks, 10
computeBounds, 12
computeStat, 15
contMC, 17
critVal, 19
jsmurf, 30
jumpint, 32
monteCarloSimulation, 34
MRC, 36
neighbours, 41
sdrobnorm, 46
smuceR, 47
stepblock, 49
stepbound, 51
stepcand, 52
stepFit, 54
stepfit, 57
steppath, 60
stepR-package, 2
stepsel, 62
testSmallScales, 63
transit, 65
∗ **package**
stepR-package, 2
∗ **ts**
dfilter, 26
.testSmallScales *(*testSmallScales*)*, 63
[.bounds *(*bounds*)*, 9
[.data.frame, *50*, *59*
[.stepblock *(*stepblock*)*, 49
[.stepfit *(*stepfit*)*, 57
[[.steppath *(*steppath*)*, 60

AIC, *59*, *61*
as.integer, *46*
assign, *9*, *23*, *37*
attr, *50*, *59*, *62*
attribute, *21*, *22*
attributes, *35*

bessel, *8*
BesselPolynomial, *3*, 8, *27*
bounds, *3*, 9, *31*, *48*, *51*, *52*, *62*

character, *23*
chi *(*MRC*)*, 36
class, *27*
compareBlocks, *3*, *4*, 10
computeBounds, *3*, *4*, *9*, *10*, 12, *16*, *25*
computeStat, *3*, *4*, *14*, 15, *35*, *42*, *45*
confband, *31*, *48*, *52*, *56*
confband *(*jumpint*)*, 32
connection, *24*
connections, *24*
contMC, *3*, *11*, 17
convolve, *27*
critVal, *3*, *4*, *12–14*, *16*, 19, *35*, *45*, *54–56*, *64*

data.frame, *10*, *11*, *13*, *15*, *18*, *33*, *50*
dbinom, *28*
dfilter, *3*, *8*, *18*, 26, *28*, *31*
diff, *46*, *47*
Distributions, *28*, *43*
dnorm, *28*
dpois, *28*

environment, *23*, *25*
exists, *23*

family, *3*, *9*, *10*, *18*, *27*, 28, *31*, *47*, *48*, *50*, *51*, *53*, *59*, *61*, *63*, *65*
filter, *27*
findInterval, *41*
fitted, *59*
fitted.stepfit *(*stepfit*)*, 57
function, *37*

get, *23*
getCacheRootPath, *23*

integer, *18*, *28*, *34*
interval system, *42*
intervalSystem, *3*, *4*, *13–16*, *20*, *25*, 29, *34*, *35*, *55*, *56*
intervalsystem *(*intervalSystem*)*, 29
is.element, *41*

jsmurf, *3*, *4*, *18*, 30, *38*, *51*, *52*, *66*
jumpint, *31*, 32, *48*, *52*, *56*

kMRC.pvalue *(*MRC*)*, 36
kMRC.quant *(*MRC*)*, 36
kMRC.simul *(*MRC*)*, 36

length.steppath *(*steppath*)*, 60
lines, *33*, *50*
lines.confband *(*jumpint*)*, 32
lines.stepblock *(*stepblock*)*, 49
lines.stepfit *(*stepfit*)*, 57
list, *15*, *18*, *21–24*, *27*, *60*, *61*, *64*
loadCache, *23*
logical, *10*, *31*, *33*, *37*, *48*, *51*, *52*, *55*, *59*
logLik, *59*, *61*
logLik.stepfit *(*stepfit*)*, 57
logLik.steppath *(*steppath*)*, 60
lowpassFilter, *42*

match, *41*
matrix, *18*
monteCarloSimulation, *3*, *12–14*, *16*, *19*, *21*, *23*, *25*, 34, *36*, *38*, *42*, *45*, *54*, *56*
MRC, *3*, 36
MRC.1000, *3*, 39
MRC.asymptotic, *3*, *31*, 40, *48*
MRC.asymptotic.dyadic, *3*, 40
MRCoeff *(*MRC*)*, 36

neighbors *(*neighbours*)*, 41
neighbours, *3*, 41

Normal, *27*
numeric, *10*, *18*, *27*, *28*, *39*, *40*, *42*

order, *14*

par, *33*
parametricFamily, *3*, *4*, *12–16*, *20*, *21*, *25*, *28*, *29*, *34*, *35*, 41, *45*, *47*, *55*, *56*
parametricfamily *(*parametricFamily*)*, 41
penalties *(*penalty*)*, 44
penalty, *3*, *4*, *14*, *16*, *20*, *25*, *34*, *35*, *42*, 44, *56*
plot, *50*
plot.default, *50*
plot.stepblock *(*stepblock*)*, 49
plot.stepfit *(*stepfit*)*, 57
points, *33*
points.jumpint *(*jumpint*)*, 32
polyroot, *8*
print.dfilter *(*dfilter*)*, 26
print.stepblock *(*stepblock*)*, 49
print.stepfit *(*stepfit*)*, 57
print.steppath *(*steppath*)*, 60

quantile, *37*, *38*

R.cache, *4*, *13*, *23*, *56*
RDS, *4*, *13*, *23*, *24*, *56*
residuals, *59*
residuals.stepfit *(*stepfit*)*, 57
rnorm, *42*, *43*

saveCache, *23*
sd, *47*
sdrobnorm, *3*, *4*, *28*, *31*, *42*, *43*, 46, *48*, *65*, *66*
set.seed, *34*
smuceR, *3*, *38–40*, 47, *51*, *52*
step, *50*
stepblock, *3*, *10*, *11*, *18*, *33*, 49, *59*
stepbound, *10*, *18*, *31–33*, *38*, *48*, 51, *59*, *63*, *66*
stepcand, *3*, *41*, *51*, 52, *58*, *60*, *61*
stepFit, *2–4*, *14*, *16*, *25*, *47*, *48*, 54
stepfit, *3*, *11*, *15*, *31*, *32*, *48*, *50*, *52*, *53*, *56*, *57*, *61*, *63*, *65*, *66*
steppath, *3*, *18*, *53*, *59*, 60, *62*, *63*, *66*
steppath.stepcand, *53*
stepR *(*stepR-package*)*, 2
stepR-package, 2
stepsel, *3*, *38*, *52*, *59*, 62

testSmallScales, 63
thresh.smuceR *(*smuceR*)*, 47
transit, *3, 4*, 65

vector, *23*

warning, *12*, *20*, *34*, *55*, *64*